Basic APPLE™ BASIC

JAMES S. COAN
author of Basic BASIC

HAYDEN

Basic APPLE BASIC™

Basic APPLE™ BASIC

JAMES S. COAN

HAYDEN BOOK COMPANY, INC.
Rochelle Park, New Jersey

Library of Congress Cataloging in Publication Data

Coan, James S.
Basic Apple BASIC.

Bibliography: p.
Includes index.
1. Apple II (Computer) — Programming. 2. Basic (Computer program language) I. Title.
QA76.8.A662C6 1982 001.64'24 82-11737
ISBN 0-8104-5626-5

Apple is a trademark of Apple Computer Co., Inc., and is not affiliated with Hayden Book Co., Inc.

Copyright © 1982 by HAYDEN BOOK COMPANY, INC. All rights reserved. No part of this book may be reprinted, or reproduced, or utilized in any form or by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying and recording, or in any information storage and retrieval system, without permission in writing from the Publisher.

Printed in the United States of America

1 2 3 4 5 6 7 8 9 PRINTING

82 83 84 85 86 87 88 89 90 YEAR

Preface

The Apple II and the Apple II Plus are very popular among the microcomputers available today. With an Apple and a conventional TV set we have a powerful, genuine computing machine. If the TV is a color set, then we have the world of low- and high-resolution color graphics at our fingertips. Add a printer, and we can get written results for distribution and filing. Add to this one or more disk drives, and the result is a machine capable of handling important business record-keeping functions. An Apple can be used to perform all the computing for a small business. Apples are found scattered throughout various departments even in very large companies that also require tremendous mainframe computers for massive computing functions. Small, low-cost microcomputers may be used to free individuals and departments within these large companies from dependence upon computer-department timetables and budgets.

The range of programs on the market is tremendous. Pure passive entertainment, games and simulations offering user interaction through the keyboard and joysticks, income tax preparation, receivables, payables, general ledger, extensive financial planning, and word processing are just a few of the current applications—and don't think that the mainframes aren't used for some games and pure entertainment.

One of the exciting developments with the advent of the affordable microcomputer is that ordinary citizens may create their own programs. The Apple offers two versions of the popular easy-to-learn BASIC language. Apple Integer BASIC is available for tasks that do not require calculations outside the range of −32767 to 32767. For decimal calculations and very convenient high-resolution graphics, Applesoft BASIC is the correct choice.

Regrettably, computer programming has acquired a mystique it doesn't deserve. Of course, professional programmers are highly skilled people. Racing-car drivers are also highly skilled people. That doesn't seem to present a barrier to learning to drive a car for the average person. And learning to drive a car doesn't suggest that one aspires to be a racing-car driver. Anybody who can manage a checking account can write a computer program. Many important and useful programs are written

without any more mathematics than addition, subtraction, multiplication, and division. If your problem can be solved on the computer and you understand it well enough to solve it using pencil and paper, then you can probably write a computer program to solve it, too. While much of programming is mathematically oriented, an effort has been made to include topics and develop programming ideas that do not require advanced mathematics.

This book is suitable for use as a textbook in schools and colleges. However, it is equally appropriate for use by the individual who wishes to learn programming in BASIC on an Apple.

The approach in this book is to begin with short complete programs and then carefully and gently build them into larger programs that solve larger problems. Over 80 distinct programs are presented as examples. Each new capability or organization of capabilities is presented to create a desired effect in a program. Generally, details are introduced in the context of the new effect on a program. Even though some of the topic headings appear to be oriented toward the BASIC language, each feature occurs at a point where it fits to solve a problem. Having those topic headings will help the reader using this book as a reference after learning to program.

Programming has developed tremendously since the "early days." This book takes advantage of many of the good programming practices we have learned in that time. We always divide the program into small manageable segments. Most segments will fit on a single Apple text screen. The longer programs consist of a control routine at the beginning that handles all program management using subroutines.

Chapter 1 gets us started entering data and getting results out of the computer. Chapter 2 introduces some ideas for planning a program. Low-resolution graphics are presented in Chapter 3, and Chapter 4 contains a potpourri of BASIC features and programming techniques. Chapter 5 presents strings, and Chapter 6 covers numeric and string arrays. Chapter 7 is a collection of miscellaneous applications. Sequential and random-access files are the topics of Chapter 8. The final chapter of the book presents high-resolution graphics using Applesoft.

Each chapter is followed by a Programmer's Corner, which highlights special features or advanced programming ideas. Programmer's Corner 1 discusses immediate-mode execution. The special screen editing on an Apple is covered in Programmer's Corner 2. Programmer's Corner 3 tells how to obtain full-screen graphics, while 4 explains how to read the keyboard without using an INPUT statement. Programmer's Corner 5 reveals the character sets used by both BASICs, and 6 presents integer variables in Applesoft. A menu program is the topic of Programmer's Corner 7, and some advanced features for file handling are presented in Programmer's Corner 8. The final Programmer's Corner is an example of a shape table.

Appendix A outlines the procedure for gaining access to both BASICs. Loading and saving programs are discussed in Appendix B. Some commonly used PEEKs, POKEs, and CALLs are listed in Appendix C, and Appendix D is an index of the programs in this book. Solution programs for the even-numbered problems appear in Appendix E. A Bibliography of books and magazines follows Appendix E.

JAMES S. COAN

New Hope, Pa.

To the Reader

Learning to program a computer can be a very exhilarating experience. The thrill of seeing your first complicated idea implemented in a program is wonderful. You will be well advised to look upon the computer as something to be mastered, not as some impersonal monster that is out to do you in. Everything that the computer does is explainable and predictable. You should take care to evaluate the results that the computer produces. Do not blindly accept computer results as faultless. That is not to say that the computer is going to make many mistakes. In fact, under normal conditions, the computer will execute your instructions exactly. Mistakes in the results of a program execution are usually caused by errors in the instructions written by the programmer. Once in a great (and I mean great) while, the problem will be a "bad" memory chip or a dirty edge connector. Don't count on it. This is absolutely the remotest possible cause of faulty program behavior; it almost never happens. Strongly resist the temptation to blame anything other than your programming for incorrect or unexpected results.

Learning to program a computer is not so complicated. You will probably find that an iterative process works best. Read some of this book. Try some things on the computer. Then reread the book. Again try some things on the computer. There are certain things that you cannot possibly know without being told and some things that make sense only based on what is known so far. You will find that reading the text will help with writing the next program and that writing and executing a program will help with reading the text.

I hope that you will be stimulated by your work in programming to bring to the computer your new and exciting problems to be solved. Above all, to be successful you will have to be an active participant. Actually write programs, execute them, and then try to see how what you have learned fits into the picture of the BASIC language and programming in general.

Experiment; write programs to solve problems of interest to you. You can't do any physical damage to the computer by typing the wrong thing at the keyboard. Don't be afraid to try anything.

GOOD LUCK!

Contents

Chapter 1

PROGRAMMER'S CORNER 1

Chapter 2

PROGRAMMER'S CORNER 2

Chapter 3

Apple Graphics (Lo-Res) and Much More .. 43

PROGRAMMER'S CORNER 3

Chapter 4

Miscellaneous Features and Techniques... 59

PROGRAMMER'S CORNER 4

Chapter 5

Character Strings and String Functions 79

PROGRAMMER'S CORNER 5

Chapter 6

PROGRAMMER'S CORNER 6

Chapter 7

PROGRAMMER'S CORNER 7

Chapter 8

PROGRAMMER'S CORNER 8

Chapter 9

Appendix D

Appendix E

Chapter 1

Introduction to BASIC on an Apple Computer

A program is a set of instructions that causes the computer to perform in a predictable way. The process of writing those instructions for the computer is called *programming.* We can write programs to do an amazing variety of things. The Apple can do a wide range of arithmetic operations. It can be programmed to play music on its built-in speaker or to draw graphs, in color no less. Paddles or joysticks can be used to provide a continuous range of responses by moving a lever or rotating a dial. We can even write programs to respond to a light pen drawing on a TV screen. There are many ways in which this computer is being used to help students learn subject matter unrelated to computers. This same computer can be used to keep track of all kinds of data necessary in the operation of a small business.

Every instruction used in programs has its own precise definition, and the total collection of these instructions is called a *computer language.* Each instruction of the language has a form associated with it. This form is called the *instruction syntax.* The syntax of each instruction that we enter into the computer must be one of those which the computer "recognizes." For example, the computer will reject the instruction QUIT, whereas it will find END perfectly acceptable and will indeed end upon encountering the END instruction. Even though QUIT and END have similar meanings in English, the computer won't behave that way. Words that make up the language are called *keywords.* END is a BASIC keyword.

The Apple is capable of working with several languages. The two languages presented in this book are Applesoft and Apple Integer BASIC. Both languages resemble the BASIC that was developed at Dartmouth

College by John G. Kemeny and Thomas E. Kurtz. BASIC is designed so that people ranging from the rank amateur to the advanced engineer can quickly and easily write programs pertinent to problems of their own interest.

As the name implies, Apple Integer BASIC is limited to integer arithmetic in the range from −32767 to 32767 and does not provide for working with decimal values easily. Applesoft, on the other hand, provides up to nine decimal digits in arithmetic calculations. There are other differences, which we will discuss as they come up. In this book, when we refer to BASIC, the discussion applies to both Apple BASICs. The terms "Applesoft" and "Integer BASIC" will be used to designate features limited to one or the other Apple language. Both languages incorporate features that enable us to easily use the special features of the Apple computer.

Appendix A discusses the various options for obtaining access to Applesoft and Apple Integer BASIC.

1-1.1...Getting Started in Applesoft

There are a number of things that we need to know, all at once, to get going. After this initial burst of information, we can introduce things in smaller doses. So, here we go!

```
]NEW

]100 PRINT "HERE IS AN EXAMPLE"

]110 PRINT "OF A PROGRAM IN"

]120 PRINT "APPLESOFT."

]
```

Program 1-1. Our first Applesoft program.

There you have it: our first program in Applesoft. Of course, each line we type must be followed by pressing the RETURN key. Every time we typed the RETURN key the Apple responded with the right square bracket. The right square bracket (]) is the Apple's signal to us that we are working with Applesoft. This is called the Applesoft prompt. The first thing that we did was to prepare the Apple for a new program with the instruction NEW. NEW erases any BASIC program in the Apple's memory. Naturally, you should never type NEW unless you really mean it. The old program is not recoverable. Appendix B deals with the subject of saving programs for future use.

Our program consists of three statements. Each statement is labeled with a line number. An Applesoft line number may be any integer from Ø to 63999. After the second], we typed:

```
1ØØ PRINT "HERE IS AN EXAMPLE"
```

and pressed the RETURN key. The Apple responded by showing a blank line and then another right bracket. We then typed the next two lines in the same manner. Each of these three statements is an example of the PRINT statement. When the program is RUN, each PRINT statement is an instruction to the computer that something is to be printed out to the screen of the TV monitor and/or on the paper in a printer.

```
]RUN
HERE IS AN EXAMPLE
OF A PROGRAM IN
APPLESOFT.

]
```

Figure 1-1. Execution of Program 1-1.

Next, we typed RUN and pressed the RETURN key. In this case, as with the NEW instruction, we did not give a line number to the instruction. The presence of a line-number label means that the current line is to be stored for later use by the computer. The absence of a line-number label means that the computer will immediately process whatever is on the line as an instruction. The RUN instruction causes the computer to process the instructions of the program stored in the computer's memory. That is what is meant by "running a program." The RUN instruction may also be followed by a line number that names a line in the program where the run should begin. For example, to get our little program to display "APPLESOFT", simply enter RUN 12Ø.

The result of processing the instructions of the program stored in the computer's memory is that the three PRINT statements cause whatever is enclosed within quotes to be printed on the monitor.

When the Apple runs out of instructions in the stored program, it simply displays the] prompt and politely waits for us to tell it what to do next. If we now type RUN again, the Apple will display the same three-line message on the monitor.

Note the difference between the letter "oh" and the digit zero. The Apple uses an oval with a slash through it for the digit zero and an open oval for the letter "oh." You might just type "ohs," zeros, and eights so that you can study them on the monitor. You will find the zero key between "9" and ":" in the top row of keys, while the "oh" is in the *second* row from the top between the "I" and "P" keys.

The following is a way to change the displayed message:

```
]11Ø PRINT "OF A PROGRAM"

]115 PRINT "WRITTEN IN"

]
```

Figure 1-2. Changing Program 1-1.

We have changed line 11Ø by retyping it. We have inserted a new line, numbered 115. By choosing a line number between two existing line numbers, we have told the computer that we want line 115 to be processed after line 11Ø and before line 12Ø. It is a good idea to allow intervals in your line numbering. The computer will always arrange the lines of the program in increasing order. Now, if we tell the computer to follow the instructions of the new program, we get:

```
]RUN
HERE IS AN EXAMPLE
OF A PROGRAM
WRITTEN IN
APPLESOFT.

]
```

Figure 1-3. Execution of the modified Program 1-1.

We call the process of carrying out the instructions of program statements *execution*. Thus, when we type RUN, we are telling the computer to "execute" the program.

At this point, we have a program that we have created in two distinct steps. We first entered three lines, and some time later we entered two lines. One of those two lines replaced a line of the earlier program, and the other added a new instruction line. The resulting program contains four lines.

It is now desirable to look at the program in its entirety by using the LIST instruction.

```
]LIST

1ØØ  PRINT "HERE IS AN EXAMPLE"
11Ø  PRINT "OF A PROGRAM"
115  PRINT "WRITTEN IN"
12Ø  PRINT "APPLESOFT. "

]
```

Figure 1-4. Demonstrate LIST.

The instructions NEW, RUN, and LIST are commonly called *commands* because they are used to command the computer to manipulate the program as an entity rather than perform a program instruction.

What happens when we make typing errors? That depends upon the error. If we type LOST instead of LIST, Applesoft will make a bell-like sound and display the following message:

```
?SYNTAX ERROR
```

No harm has been done; merely correct the request and proceed. If we type

```
100 PRIMT A
```

instead of

```
100 PRINT A
```

nothing will happen until the statement is executed. At that time, execution will cease and Applesoft will emit its bell-like sound and display the following message:

```
?SYNTAX ERROR IN 100
```

We can look at that single statement by using an extension of the LIST command.

```
LIST 100
```

will display only line 100 of our program, if it exists. LIST 100,200 will display all of the lines in our program from 100 to 200, inclusive. Now retype the line and reexecute the program. If we type

```
100 PRINT B
```

instead of

```
100 PRINT A
```

we have a different kind of error, which the computer will never find for us. The value of B will be displayed where we expected to see the value of A. It is important to evaluate our results for correctness.

.... SUMMARY

A computer language is a defined set of instructions that have specific meaning to the computer. In BASIC on an Apple, each instruction of a

program begins with a line number. The PRINT statement is used to display a message on the computer monitor.

The NEW command prepares BASIC for a new program, the RUN command causes the Apple to carry out the instructions of the program stored in its memory, and the LIST command displays the stored program on the monitor. LIST 10Ø displays the line numbered 10Ø, while LIST 10Ø,20Ø displays all lines in the interval from 10Ø to 20Ø including 10Ø and 20Ø.

1-1.2...Getting Started in Apple Integer BASIC

This section is devoted to describing the differences between Applesoft and Apple Integer BASIC as they apply to the material presented so far. So, you should read Section 1-1.1 even if you are working only with Integer BASIC.

The Apple Integer BASIC prompt is the "greater than" sign (>). It is this symbol that reminds us that we are in Integer BASIC. Integer BASIC does not insert a blank line between the lines that we type. Integer BASIC requires an END statement, which signifies the end of the execution of the program. So, a program similar to our first example above would look like Program 1-2.

```
>NEW

>100 PRINT "HERE IS AN EXAMPLE"
>110 PRINT "OF AN APPLE"
>120 PRINT "INTEGER BASIC"
>130 PRINT "PROGRAM."
>130 END
>
```

Program 1-2. Our first Integer BASIC program.

```
>RUN
HERE IS AN EXAMPLE
OF AN APPLE
INTEGER BASIC
PROGRAM.

>
```

Figure 1-5. Execution of Program 1-2.

The line-number range is from Ø to 32767. Integer BASIC programs are listed slightly differently by the LIST command.

```
>LIST
  100 PRINT "HERE IS AN EXAMPLE"
  110 PRINT "OF AN APPLE"
  120 PRINT "INTEGER BASIC"
  130 PRINT "PROGRAM."
  140 END

>
```

Figure 1-6. Demonstrate LIST in Integer BASIC.

Note that the listing is indented. Integer BASIC lists programs with the first character of each statement in the seventh column. This allows room for five-digit line numbers.

Some of the differences are minor and have no effect on the statements used to achieve a desired result. Other differences are more serious. For example, Integer BASIC programs must execute an END statement to avoid the embarrassment of the following error message:

```
*** NO END ERR
```

accompanied by a bell.

What happens when we make typing errors under the influence of Integer BASIC? If we type LOST instead of LIST, Integer BASIC will make a bell-like sound and display the following message:

```
*** SYNTAX ERR
```

No harm has been done; merely correct the request and proceed. If we type

```
100 PRIMT A
```

instead of

```
100 PRINT A
```

we will get the same gentle message. Unlike Applesoft, Integer BASIC checks each program line for syntax at the time that we actually type it. Thus, we cannot be taken by surprise with syntax-error messages during program execution.

1-2...Printing Messages

There were no long program statements in the last 2 sections. The Apple monitor screen is 40 characters wide. That can be something of a

limitation. However, we will very quickly get used to it. When typing program statements that are longer than 40 characters, just keep on typing. The Apple will take care of everything.

```
]100 PRINT "HERE IS AN EXAMPLE OF AN APP
LESOFT PROGRAM."

]RUN
HERE IS AN EXAMPLE OF AN APPLESOFT PROGR
AM.

]LIST

100  PRINT "HERE IS AN EXAMPLE OF
      AN APPLESOFT PROGRAM."

]
```

Figure 1-7. Demonstrate the 40-column display screen.

The Apple will take care that no characters are lost. In the interest of making the results of printing readable, we should plan ahead so that the Apple doesn't break the line in an awkward place during program execution. To print the message of the above program, we might prefer the following:

```
]100 PRINT "  HERE IS AN EXAMPLE OF AN A
PPLESOFT"
]110 PRINT "PROGRAM."

]RUN
  HERE IS AN EXAMPLE OF AN APPLESOFT
PROGRAM.

]LIST

100  PRINT "  HERE IS AN EXAMPLE
     OF AN APPLESOFT"
110  PRINT "PROGRAM."

]
```

Figure 1-8. Planning messages on the Apple screen.

Now we can easily read the message. With a little practice, printing messages will become second nature to us.

To print the same message using Integer BASIC, simply add an END statement to the program. We might also change the wording of the mes-

sage to indicate Integer BASIC. The LISTing in Integer BASIC will break the long line 10Ø in a slightly different spot, but that has no bearing on what we type in or how we type it.

We may make messages as long or as short as we like. A program could consist of hundreds of PRINT statements. All programs should have at least one PRINT statement. How would we know what the program does if it displays no message?

Everything that we do on the Apple will use uppercase letters. That is the way Apple does it. Special programs and accessories are available to utilize both upper- and lowercase letters. However, their use will have no effect on our ability to learn to write programs.

1-3.1...Doing Calculations in Applesoft

Now that we know how to display messages, we might like to have something for the messages to talk about. The Apple's ability to perform calculations is readily available to us.

```
   10Ø  PRINT "THE NUMBERS ARE:"
   105  PRINT "234.56 AND 439Ø1"
   11Ø  PRINT "THE SUM IS"
*  12Ø  PRINT 234.56 + 439Ø1
   13Ø  PRINT "THE DIFFERENCE IS"
*  14Ø  PRINT 234.56 - 439Ø1
   15Ø  PRINT "MULTIPLY THEM"
*  16Ø  PRINT 234.56 * 439Ø1
   17Ø  PRINT "NOW DIVIDE"
*  18Ø  PRINT 234.56 / 439Ø1
```

Program 1-3. Calculations in Applesoft.

Here we have a program that performs addition, subtraction, multiplication, and division of two numbers. As you can see in lines 12Ø, 14Ø, 16Ø, and 18Ø, Apple uses +, −, *, and / as the symbols for these arithmetic operations. Figure 1-9 is an execution of this program.

```
]RUN
THE NUMBERS ARE:
234.56 AND 439Ø1
THE SUM IS
44135.56
THE DIFFERENCE IS
-43666.44
MULTIPLY THEM
1Ø297418.6
NOW DIVIDE
5.34293Ø69E-Ø3
```

Figure 1-9. Execution of Program 1-3.

Notice that the product displays nine digits. This is the maximum precision in Applesoft. That is not to say that all answers are accurate to nine digits. Sometimes the computer has to round things off. So, it is up to you to verify the accuracy of computed results. In addition to evaluating the accuracy of the computations that the computer carries out, you will need to know the accuracy of the numbers that you give the computer to work with.

For the division problem of our program, something else interesting happens. We get 5.34293Ø69E-Ø3 as the answer. This is another way of writing .ØØ534293Ø69, which takes 11 digits to express. Applesoft uses scientific notation for displaying very small and very large numbers. In Applesoft any value less than .Ø1 or greater than 999999999.1 will be displayed in this manner.

Applesoft limits numbers to a range of from −1E38 to +1E38. That should be entirely adequate for our needs for some time to come.

The following is a little program to demonstrate a few numbers printed in scientific notation:

```
100  PRINT "EXAMPLES OF SCIENTIFI
     C NOTATION"
120  PRINT ".0001","= ";.0001
125  PRINT ".00058293","= ";.0005
     8293
130  PRINT ".00123456789","= ";.0
     0123456789
140  PRINT "1234567890","= ";1234
     567890
150  PRINT "3939382827347456= ";3
     939382827347456
```

Program 1-4. Demonstrate scientific notation.

In Program 1-4 we have used single PRINT statements to print two items on one line. Look at line 12Ø. There you will see that the first thing to be printed is enclosed in quotes. Then a comma appears. A comma as used here is an instruction to the computer to display the next character beginning in the next field. Applesoft divides each line into three fields. The 1st field consists of columns 1 through 16. Columns 17 through 32 make up the 2nd field, and the remaining 8 columns make up the 3rd field. The 2nd field is used if there is something displayed in the 1st field and column 16 is empty, while the 3rd field is used if columns 24 through 32 are vacant. The 1st field is used if we are starting a new line or if the 3rd field of the previous line is filled or unusable. The next thing we see in line 12Ø of the program above is more data enclosed in quotes for printing. Then we see a semicolon. A semicolon is used to separate items in a PRINT statement when we want the computer to keep printing without

skipping any columns on the screen. We have used commas and semicolons to separate different items in a program. Symbols used in this way are called *delimiters*.

Now let's look at the display produced by the above program.

```
]RUN
EXAMPLES OF SCIENTIFIC NOTATION
.0001            = 1E-04
.00058293        = 5.8293E-04
.00123456789     = 1.23456789E-03
1234567890       = 1.23456789E+09
3939382827347456= 3.93938283E+15
```

Figure 1-10. Execution of Program 1-4.

We will indeed get used to the 40-character screen on the Apple. However, we are not limited to 40 characters on the printed page. Therefore, we will present most of our program listings in a wider format. If you type exactly what is displayed in the programs of this book, BASIC will take care of the rest. We present here Program 1-4 reformatted without any line breaks.

```
100  PRINT "EXAMPLES OF SCIENTIFIC NOTATION"
120  PRINT ".0001","= ";.0001
125  PRINT ".00058293","= ";.00058293
130  PRINT ".00123456789","= ";.00123456789
140  PRINT "1234567890","= ";1234567890
150  PRINT "3939382827347456= ";3939382827347456
```

Program 1-4a. Demonstrate program listings without line breaks.

1-3.2...Doing Calculations in Integer BASIC

Integer BASIC has the restriction that all numbers represented within a program must be in the range from −32767 to +32767. If ever a number outside this range is encountered, you will be notified by the following message:

```
*** >32767 ERR
```

This is the message you get even if the value detected is less than −32767. It is worth noting that this is also the message you get for trying to divide by 0.

It is important to realize that division can never produce decimal values. Integer BASIC simply ignores any decimal part of the result of a division process. While this may seem like a serious limitation, we will be

developing program techniques to minimize the impact of the restriction. We will even see how to do infinite-precision arithmetic. A surprisingly large number of applications never require decimal values.

Program 1-5 is a simple program to demonstrate addition, subtraction, multiplication, and division.

```
100 PRINT "SOME APPLE INTEGER BASIC CALCULATIONS"
110 PRINT "109 + 23 = ";109+23
120 PRINT "109 - 23 = ";109-23
130 PRINT "109 * 23 = ";109*23
140 PRINT "109 / 23 = ";109/23
999 END
```

*Program 1-5. Demonstrate +, −, *, and / in Integer BASIC.*

A semicolon is used to separate items in each of the PRINT statements of this little program. A semicolon used in this way allows the next character to be printed in the very next space on the screen without inserting any spaces. When the 40th space is filled, the next character is printed at the beginning of the next line.

When a comma appears as a separator in a PRINT statement, Integer BASIC divides the 40-column display screen into 5 fields of 8 characters. The next character displayed will go in the next available 8-character field. That may cause the next item to be displayed on the next line. Again, characters used to separate items on a line in a program are called delimiters.

```
>RUN
SOME APPLE INTEGER BASIC CALCULATIONS
109 + 23 = 132
109 - 23 = 86
109 * 23 = 2507
109 / 23 = 4
```

Figure 1-11. Execution of Program 1-5.

1-4...Numeric Variables

We can do some interesting things with what we know at this point, but the real power of the computer begins to emerge when we can save the results of calculations without having to recalculate.

A variable may be thought of as a pigeonhole or a mailbox in which we may save the value of an intermediate result as a computer program goes about solving our problem for us. We establish an hourly wage and save it

in "W1". Then we get the number of hours worked and save the value in "N". Next, we might find the net pay by multiplying W1 by N and then save it in "N9". Or we might want to take the average of some numbers. Program 1-6 uses numeric variables to do just that.

```
100  LET S1 = 34 + 45 + 65 + 89 + 91 + 56
110  LET N1 = 6
120  LET AV = S1 / N1
130  PRINT "AVERAGE = ";AV
140  END
```

Program 1-6. Calculate a simple average.

If we want to calculate an average for a different set of values, we need only retype lines 100 and 110 of this simple program.

We have used 3 variables in this program: S1, N1, and AV. We are free to choose a wide variety of names for variables. Both Apple BASICs allow any letter followed by digits or letters. Apple Integer BASIC allows up to about 100 characters, while Applesoft allows up to 238 characters. However, Applesoft uses only the first 2 characters of the variable name to distinguish between 2 variables. Thus, in Applesoft, OLDNUMBER and OLDSCORE will be the same variable. You should avoid names like this in Applesoft. One method for avoiding most trouble is to limit variable names to 1 letter, 2 letters, or a letter followed by a digit. While variable names like WAGES, NETPAY, PAYCHECK, PAYRATE, and NETTAXES, are descriptive, we run the risk of naming ambiguous variables in Applesoft. And in either language, very long variable names are going to push program statements over onto multiple lines. Just try typing a 100-character variable name the same way twice in a row.

BASIC keywords are reserved for use by BASIC itself. When BASIC encounters a reserved word as a variable name or as part of a variable name it may be interpreted as an instruction instead of as a variable name. It is best to steer clear of any keywords when selecting variable names. Errors caused by incorrectly using reserved words for variable names can be tough to find. NEW, LIST, RUN, PRINT, and END are reserved words. In Integer BASIC we can use some of the BASIC keywords as variable names, but it isn't worth the trouble to remember which ones we can and can't use. Just don't use them at all.

Each of the statements 100, 110, and 120 of Program 1-6 is an example of the *assignment statement* in BASIC. The effect of statement 100 is that the Apple will calculate the sum of the 6 numbers shown there and store it in the slot labeled "S1". Line 110 causes the value 6 to be stored in a pigeonhole labeled "N1". Line 120 causes the computer to divide the value found in the slot labeled "S1" by the value found in the slot labeled "N1" and place the result in a slot labeled "AV".

```
]RUN
AVERAGE = 63.3333333
```

Figure 1-12. Execution of Program 1-6.

We note that while Applesoft gave us 63.3333333, Integer BASIC would have produced 63.

The assignment statement in BASIC may take one of two forms.

```
100  LET X=15
```

and

```
100 X=15
```

are functionally equivalent. In practice, most programmers drop the use of LET. However, many beginners find it helpful to include the LET keyword while learning BASIC.

1-5...The INPUT Statement

BASIC includes the INPUT statement as one means of providing data for a program to work on. When BASIC encounters an INPUT statement, it causes the computer to wait for data to be typed at the keyboard.

When the statement

```
200  INPUT X
```

executes, it will display a question mark as the signal to us that we are to type in a single number.

```
200  INPUT X,Y,Z
```

will also display a question mark. However, we have provided for three values to be entered, and the computer will insist on getting three.

Suppose we enter only one. Applesoft will gently prod us by displaying two question marks repeatedly until we have entered the proper amount of data. Integer BASIC will simply display one question mark repeatedly until enough data has been entered.

Suppose we enter too much data. Applesoft will quietly display the following message:

```
?EXTRA IGNORED
```

and proceed with the rest of the program. Integer BASIC does not check for extra values.

Suppose we just hit the RETURN key or type a letter instead of a number. The following is Applesoft's silent treatment:

```
?
?REENTER
?5,A,2
?REENTER
?
```

By REENTER, Applesoft means reenter the entire line. The "5" typed in this example is not retained. So, all three values must be entered on the next try. In Integer BASIC it looks like the following:

```
?
*** SYNTAX ERR
RETYPE LINE
?5,A,3
*** SYNTAX ERR
RETYPE LINE
?
```

The *** SYNTAX ERR display is accompanied by a bell.

Suppose we have the following record of gasoline purchases for a brand-new car:

GALLONS	ODOMETER
19.3	23Ø.3
12.7	456.7
17.7	7Ø9.4
11.1	895.5
13.8	1131.6

We want a program that will calculate the mileage for each tankful of gasoline. We have been careful to fill the tank each time. Since we do not know whether the tank was full when we got the car, we should discard the figure for the 1st purchase. What we do know is that 12.7 gallons took us 226.4 miles, 17.7 gallons took us 252.7 miles, etc. Program 1-7 asks the right questions and does the miles-per-gallon calculation for us.

```
100  PRINT "FIRST READING";
110  INPUT M1
195  PRINT
200  PRINT "GALS,READING";
210  INPUT GA,M2
220 MI = M2 - M1
230 MG = MI / GA
240  PRINT MG;" MPG"
250 M1 = M2
260  GOTO 195
```

Program 1-7. Calculate gasoline mileage.

Note the use of the semicolon at the end of lines 100 and 200. This enables us to compose a single line of display on the screen from several lines in a program. Thus the question mark displayed by the INPUT statement at line 110 will appear immediately following the G in READING from line 100, and the question mark displayed by the INPUT statement at line 210 will appear immediately following the G in READING from line 200.

It is always a good idea to display a label for an INPUT request. We may know right now what that question mark means, but nobody else will know, and next week we probably won't remember.

Since we want the number of miles traveled, the program must subtract the previous reading from the current one. This is done in line 220. MI is the miles traveled, GA is the number of gallons used, and MG is the number of miles per gallon. Once the computer has calculated the number of miles traveled, the current reading becomes the previous reading for the next item of data to be entered. This is the purpose of line 250. Line 195 is referred to as a blank PRINT. A blank PRINT will be displayed as a blank line. We can use this to adjust the spacing for better legibility.

.... GOTO

We must introduce the GOTO statement at this point. The GOTO 195 you see at line 260 is an instruction to the computer to execute the statement numbered 195 next in sequence. In this way, we are able to control the order in which BASIC executes the statements of a program.

```
]RUN
FIRST READING?230.3

GALS,READING?12.7,456.7
17.8267717 MPG

GALS,READING?17.7,709.4
14.2768362 MPG

GALS,READING?11.1,895.5
16.7657658 MPG

GALS,READING?13.8,1131.6
17.1086956 MPG

GALS,READING?

BREAK IN 210
]
```

Figure 1-13. Execution of Program 1-7.

Clearly the value 17.8267717 shown in Figure 1-13 is more precise than 12.7, 456.7, or 230.3, but it is not more accurate. We may safely say that

we got about 17.8 miles per gallon for the 1st calculation. The results of a calculation can never be more accurate than the data. Soon we will learn how to round off results to any desired precision.

What did we do to deserve the message

```
BREAK IN 210
```

We can halt execution of a program by entering CTRL-C in response to an INPUT request. Applesoft rings the bell and prints the "BREAK IN . . ." message. Integer BASIC simply displays its prompt (>). For our first program using INPUT we have chosen to halt execution by entering CTRL-C. This is OK for programmers, but soon we will see better ways to exit our programs. We should not require others who will be using our programs to use CTRL-C in this way.

1-6.1...READ . . . DATA in Applesoft

There are numerous ways to provide programs with data. We have gotten numbers into our programs by including them in PRINT statements, by assigning values to variables with the assignment statement, and by programming with the INPUT statement. Now we add READ and DATA to the list for Applesoft.

The READ statement assigns values to variables by using a DATA statement as the source. READ and DATA are always coordinated to solve the problem at hand. It is not sensible to have one without the other. Let's simply convert the INPUT-based program on gasoline mileage to an equivalent program using READ and DATA. In this case the data will not be entered from the keyboard during execution, so we display the gallons and miles traveled along with the miles-per-gallon figure. The logic here is identical to the logic of our Program 1-7.

```
90  PRINT "GAL.  MILES  MPG."
100  READ M1
200  READ GA,M2
210 MI = M2 - M1
220 MG = MI / GA
230  PRINT GA;"  ";MI;"  ";MG
240 M1 = M2
250  GOTO 200
900  DATA 230.3
902  DATA 12.7, 456.7
904  DATA 17.7, 709.4
906  DATA 11.1, 895.5
908  DATA 13.8, 1131.6
```

Program 1-8. Program 1-7 with READ . . . DATA.

Note that we have arranged the DATA so that the numbers are grouped to look like the table of values first presented. The computer doesn't care how many or how few lines we use for DATA. The important point is that the values in DATA statements be in the correct order. We may arrange the DATA so that it is well organized for humans to read. Since the computer will never be confused, we should take care to make things better for us.

```
]RUN
GAL.   MILES   MPG.
12.7   226.4   17.8267717
17.7   252.7   14.2768362
11.1   186.1   16.7657658
13.8   236.1   17.1Ø86956

?OUT OF DATA ERROR IN 2ØØ
```

Figure 1-14. Execution of Program 1-8.

We do indeed get the expected calculation results. However, we have also triggered an Applesoft error message. Soon, we will find a couple of ways to handle the end-of-data condition in our programs.

.... SUMMARY

We have covered a lot of ground in this first chapter. Very soon, nearly everything presented here will be second nature to you. You can make your job easier by remembering the right things. Don't bother remembering exact error messages and whether or not a bell will sound along with the message. The Apple is well suited to keeping track of things like that. Your job will be simply to recognize them. Know that (>) means that you are using Integer BASIC and that (]) means that Applesoft is the current language. It matters not that Applesoft inserts spaces in program listings differently than Integer BASIC. You need to remember things like NEW, LIST, RUN, PRINT, LET, END, line numbers, GOTO, INPUT, variables, quotes, and READ . . . DATA.

Integer BASIC limits all numeric representation to the range from 32767 to −32767, while Applesoft allows numbers in the range of −1E38 to 1E38.

The PRINT statement in BASIC is used to display labels in quotes, numeric values expressed literally, and values stored in variables individually or in combination by separating items with semicolon or comma delimiters.

Variables are used in programs to retain numeric values during program execution. Variable names must begin with a letter and may consist of letters and digits intermixed after the first character. Applesoft accepts

very long variable names, but distinguishes only the first two characters. Keywords that are part of the programming language, such as LET, PRINT, and END, are not used as variable names. They are reserved for use by BASIC only.

Values may be assigned to variables using the BASIC assignment statement. The use of LET in such statements is optional. Values may also be assigned through the keyboard using INPUT statements.

Applesoft provides the companion statements READ and DATA for storing data within the program itself. Integer BASIC does not.

Execution of an END statement halts the RUN of a program. Failure to execute an END statement as the final statement in Integer BASIC will trigger an error message.

We have three commands for program manipulation thus far. RUN calls for our program to be executed. RUN 10Ø calls for our program to be executed beginning at line 10Ø. LIST displays our program or a segment of our program on the screen. NEW clears the BASIC work area for a new project.

We can halt a program waiting for INPUT by entering a CTRL-C.

Problems for Chapter 1

You should not feel that you must limit yourself to the problems offered here. As you get some programming under your belt, you should find lots of interesting problems to try on the computer. Learning to program is unique in that the computer will provide you with a measure of your success. You do not need an answer book or a teacher for that. The real joy of learning anything comes when you begin to formulate the problems, solve them, and verify that your solutions are correct all on your own (it helps to have a computer).

At this point you can write programs to print messages of all kinds, request data from the keyboard, and perform a variety of arithmetic operations.

Each problem is identified with one or more of the symbols "A", "I", and "*". An "A" indicates that the problem is appropriate for an Applesoft program. An "I" indicates that the problem is appropriate for an Integer BASIC program. An "*" indicates that the problem may be a little harder than the others. The "A" and "I" designations will be omitted where the section is limited to one language.

A 1. Write a program to display the sum of 123.45, 654, 192Ø, 114423, and .Ø1.

AI 2. Write a program to display the sum of five numbers to be supplied during execution.

A 3. Write a program to print a decimal value for 2/3.

A 4. Assign 1/3 to the variable X. Display the value of X, 3*X, and X + X + X.

A 5. Write a program to display decimal values for 1/7, 2/7, through 6/7.

A 6. Write a program to find a value for the following expression:

$$\frac{1/2 + 1/3}{1/3 - 1/4}$$

AI 7. Write a program to find the sum of the first 10 counting numbers.

A 8. Write a program to find the product of the first 10 counting numbers.

I 9. RUN Program 1-7 in Integer BASIC to get approximate values for gas mileage by multiplying both miles and gallons by 10 before entering the figures.

AI 10. Have the computer request the numerator and denominator for two fractions to be multiplied. Print the numerator and denominator of the product. This problem does not call for the computer to perform division.

AI 11. Have the computer request the numerator and denominator for two fractions to be added. Print the numerator and denominator of the sum. This problem does not call for the computer to perform division.

I* 12. In Integer BASIC the result of 7/2 is 3. We can determine the remainder by multiplying 3 by 2 and subtracting from 7. If we multiply the remainder by 10 and divide that by 2, we will get the 10ths digit of the quotient. Write a program to do this.

PROGRAMMER'S CORNER 1

Immediate Execution

We have done a variety of calculations using LET and PRINT statements in programs. Doing things this way is called *deferred execution.* Most computing is done in this way. Programs prepared for deferred execution may be saved and used over and over again.

The features of our BASIC programs may be used in a second important way. We may simply type BASIC instructions whenever a BASIC prompt is displayed. Suppose we want to know the number of hours in a year. We are not required to enter a program line number to obtain such a simple result. Simply type

```
]PRINT 365*24
```

and press the RETURN key. Instantly BASIC will execute the instruction to produce the desired value.

```
8760

]
```

We could even type a series of BASIC instructions without creating a stored program. We could find the number of hours in a year as follows:

```
]D=365

]H=24

]PRINT D*H
8760
```

Using this technique, called *immediate-mode execution,* each statement is executed immediately. It turns out that in Applesoft the keyword PRINT may be replaced with a question mark (?). This means that we may command the computer to display a result with a single keystroke instead of the five keystrokes required for PRINT. We may also use this in programs. When we call upon Applesoft to LIST the statement, it will display as PRINT.

Immediate mode may be used for several purposes. BASIC may be used as a sophisticated calculator. Detailed and complex calculations may be done quickly and easily. Whatever we enter remains on the screen for us to examine (up to 23 lines with spaces). Generally, calculators retain only a single visible number on the display. The large screen provides the opportunity to check our work. We can be more secure than with most calculators.

Immediate execution will provide a very convenient method for finding errors in deferred-execution programs. We may work through a program displaying values of selected variables until we spot one that deviates from the expected value. At that point we may even set the correct value and direct the computer to begin execution from that point with a statement such as

```
GOTO 355
```

The combination of being able to display and set values in this way will prove to be of tremendous importance for developing larger programs.

There are limits when using immediate execution to work through a program. In Applesoft everything is fine as long as we do not enter a new program statement with a line number. Doing that will result in all variables being set to zero. Integer BASIC, on the other hand, preserves the variable values all right, but under certain conditions will produce strange results. It is best not to add or remove lines of a program if we expect to try to pick up execution without a RUN command.

.... Stepping through a Program

We may use CTRL-C at any time to halt execution of a program. We may select strategic locations in our program to insert END statements. When the program stops, we may do our thing in immediate. Then we may pick up execution with the GOTO statement in immediate. Or we may use CONT in Applesoft or CON in Integer BASIC. In addition, the STOP statement may be used in Applesoft to cause a program to stop at preselected points. When Applesoft encounters a STOP statement, a message will be printed as follows:

```
BREAK IN 945
```

If we are inserting temporary STOP statements, it is very nice to know which one the program has just encountered.

Of course, certain BASIC statements make no sense entered directly from the keyboard without line numbers. Statements like READ, DATA, and INPUT are clearly intended for use in executing programs. Entering INPUT A without a line number will evoke an error message from BASIC. Which message we get depends on which BASIC we are using and whether or not we have booted a disk-operating system.

Chapter 2

Writing a Program

2-1...Planning Your Program

Computer programs are linear. That is, they define a single step at a time. Many problems brought to the computer for solution are nonlinear in nature. We would like to do many things in at least two dimensions. Many computerized processes are outlined using large charts in which each item represents a complete subsystem consisting of a whole collection of very long computer programs. We need to develop some ideas that will help us begin with the big ideas and systematically arrive at a completed project whose smallest elements are computer-program statements.

Good programming requires a plan. The planning should be completed before any program statements are written down. You should write out the entire program on paper before you sit down to type it into the computer. Major changes in program organization are easy to deal with before the program has been typed into the computer, because there is less inertia to overcome. Once a program has been typed into the computer, part of the problem becomes how to make the desired change while preserving as much as possible of what exists. While the program is still written out in longhand, such changes are much easier. Certainly, we can easily write a program to add 2 numbers without much fuss. The plan can be in our head and we can "write down" the program statements directly at the computer keyboard. But try writing a system to launch a satellite or a system to do payroll—or even a program to find all prime integers from 1000 to 2000. Good planning requires a complete under-

standing of the problem. What is known? What is the question? What will be the form of the solution? How do I get from the known information to the solution?

.... Counting on the Computer

Let's start with something simple. Let's develop a plan for getting the computer to count. This is a good first problem, since it is something we are familiar with. A thorough understanding of the problem at hand is essential for writing computer programs. It is highly unlikely that we can write a program to solve a problem we do not understand.

We usually count by starting with the number 1 and repeatedly adding 1 to get the next counting number in sequence. That is easy! There are only two ingredients here: beginning and adding 1 repeatedly.

We can begin with a statement such as

```
11Ø C1 = 1
```

But how do we add 1? The following is a way:

```
12Ø C2 = C1 + 1
125 C1 = C2
```

In mathematics, the equals sign (=) usually asserts that two expressions have the same value. However, assignment statements in BASIC use the equals sign for a special purpose. The equals sign is used to assign the result of a calculation on the right to a variable named on the left. This allows us to combine statements 12Ø and 125 above into the single statement

```
12Ø C1 = C1 + 1
```

The variable C1 contains one value prior to execution of this statement and another value following execution of this statement. The prior value is replaced by the new value. A variable may not store two values simultaneously. Even though pigeonholes and mailboxes may hold more than one item, variables cannot.

Now we need to tell the computer to repeat the work of line 12Ø over and over again. We resist any possible urge to include a statement 13Ø C1 = C1 + 1, etc. To count to 1ØØ this way would require more than 1ØØ statements. Computers are supposed to save work, not make things harder. The way to repeat the action of line 12Ø over and over again is to include the following line:

```
13Ø  GOTO 12Ø
```

This will put the computer into a loop. Now we have three lines that would indeed cause the Apple to count, beginning with 1.

```
11Ø C1 = 1
12Ø C1 = C1 + 1
13Ø  GOTO 12Ø
```

Program 2-1. First counting program.

However, we have overlooked an important ingredient. We will never know which number the computer is up to at any particular time, except when it gets to 32767 in Integer BASIC, or something above 1E38 in Applesoft. What we need here is to display each number as the computer gets to it. Therefore, we will insert a PRINT statement between lines 11Ø and 12Ø. In order for this statement to be executed every time the computer adds 1, the GOTO at line 13Ø must be changed. The loop must include the PRINT statement as shown in Program 2-2.

```
  11Ø C1 = 1
*  115  PRINT C1
  12Ø C1 = C1 + 1
*  13Ø  GOTO 115
```

Program 2-2. Counting with display.

Several comments and one warning are in order here. Program 2-2 has no natural termination. If you execute this program, it will run for a very long time. You will have to type CTRL-C, hit the RESET button, pull the plug, or wait for BASIC to overflow. In order to make this a useful counting program, we need to replace the unconditional statement 13Ø GOTO 115 with one that can make a decision. This brings us to the IF . . . THEN statement.

....IF . . . THEN

Our counting program would be more useful if we just had a way for it to terminate when some predetermined number has been reached. BASIC has the ability to alter the order in which statements are executed depending on the outcome of a decision. This is called a *conditional transfer.* Suppose we want the computer to count to 7 and quit. In this case, we want to GOTO 115 on the condition that C1 is less than or equal to 7. That is easy in BASIC:

```
13Ø  IF C1 <= 7 THEN  GOTO 115
```

Line 13Ø will do the job for us. Here "less than or equal to" is symbolized with (<=). The (<) symbol represents "less than" and the (=) symbol represents "equals."

....REM: What's It All About?

While all of our programs are clear to us at the time that we write them, it is difficult to come back to an old program and recall all of the clear

thoughts that we had way back when. BASIC offers the REM statement so that we may include REMarks as part of the program. The computer will ignore all REM statements during program execution, but it will list them along with the others in response to the LIST command. Not only will those REM statements remind us about our own old programs, but they will be invaluable to others reading our programs. No program should be considered complete without REM statements. Some programmers consider REM statements so vital to the program-development process that they write them first. Not a bad idea!

REM statements should describe the action of the program or of a segment of a program. REMarks like "LOAD Y WITH 17" actually detract from the readability of a program, whereas "INITIALIZE LOW TEMPERATURE CUTOFF" describes the function of part of a program. Our REM statement should note that the program will count from 1 to 7. And now we have a counting program to type into the APPLE and RUN.

```
100  REM * COUNTING FROM 1 TO 7
110 C1 = 1
115  PRINT C1
120 C1 = C1 + 1
130  IF C1 <  = 7 THEN  GOTO 115
999  END
```

Program 2-3. Counting from 1 to 7.

For some reason, Applesoft likes to insert two spaces between the "<" and the "=" when it lists the program.

```
]RUN
1
2
3
4
5
6
7

]
```

Figure 2-1. Execution of Program 2-3.

When using IF . . . THEN, there are six options available as follows:

< less than
<= less than or equal to
= equal to
<> not equal to

> greater than
>= greater than or equal to

These symbols are called *relational operators*. Any BASIC expression may appear on either side of a relational operator.

Counting is a process that pervades computer programming. We do it all the time. How many players? How many problems? Count the number of scores so that we may compute the average for this lab test. Count the number of lab tests so that we may compute the average for this lab run of this test. The examples go on endlessly. We might be interested in counting only the odd numbers. How would we change our counting program above to do that? That is easy—just change line 12Ø to read:

```
12Ø C1 = C1 + 2
```

Now don't forget to change the REM to reflect the new function of the program.

```
1ØØ  REM * ODD INTEGERS FROM 1 TO 7
```

Misleading REMs are terrible. The extra time spent getting the REMs right will pay off in the end. Suppose we have a problem that requires even integers. In this case, line 11Ø should set the value of C1 to start at 2 or whatever we require as the 1st even integer. Again, note that the REM should reflect the function of the program.

Our little program has four important components.

1. We initialize the counting variable.
2. Some action is programmed. In our example, we display the current value of the counter.
3. The counter is incremented.
4. We test the value of the counter to determine whether or not to loop back and repeat the programmed action.

Most counting routines are used for some higher purpose than merely displaying the current value of the counter. Suppose we have a relative who has promised to give us 5 times our age in dollars on each of our first 21 birthdays. We might like to know the total number of dollars we will have received upon reaching 21. This problem can be solved with the logic of our little counting program. Here the programmed action consists of adding 5 times C1 for each year. We will use the variable D1 for this. We initialize D1 to zero. We test for 21 rather than 7. When the IF test fails, the program should print the value of D1 with an appropriate label. The following is such a program.

```
   50  REM   TOTAL $5  EACH YEAR
         ON EACH BIRTHDAY
*  100 D1 = 0
   110 C1 = 1
*  120 D1 = D1 + 5 * C1
   140 C1 = C1 + 1
   150  IF C1 <  = 21 THEN  GOTO 120
   160  PRINT "$";D1;" AFTER 21 YEARS"
   999  END
```

Program 2-4. Birthday dollars.

Look at line 10Ø. It turns out that BASIC automatically sets the values of all variables to zero when we type the RUN command. So, for our little problem, D1 would be initialized to zero for us. However, it is good programming practice to include an assignment statement anyway. Having that statement in the program makes the meaning of line 12Ø less mysterious. When programs are LISTed the line break for REM statements sometimes makes the message hard to read. You may make them more readable by using several statements with the breaks where you want them, or after you list the program you may find that you can insert spaces so that the breaks come between words. This is something that we will get used to with a little practice.

```
]RUN
$1155 AFTER 21 YEARS
```

Figure 2-2. Execution of Program 2-4.

You are the inspector in a packaging plant. Quality control requires that for any lot to be accepted, the average weight for 5 packages selected at random must be at least 18Øg. You want to write a program that asks the right questions and accepts or rejects the lot. For this problem we have the 4 components listed above. In this case, the programmed action is a little more complex. We print a label for the INPUT request, request INPUT, and add the entered weight to a variable designated for keeping track of the total weight for the 5 packages. This can be done with the statement

```
235 T1 = T1 + WT
```

where T1 is the running total weight and WT is the weight of this package. Before we have entered any package weights, the value of T1 must be Ø. When the value of the counter has passed 5, we will calculate the average in AV. If the value of AV is less than the required 18Ø then we want a reject message. Otherwise, we want an accept message. Program 2-5 does it all.

```
   100  REM   CHECK AVERAGE PACKAGE
        WEIGHT FOR 180 GRAM MINIMUM
*  200 T1 = 0
   210 C1 = 1
   220  PRINT "WT ";C1;
   230  INPUT WT
   235 T1 = T1 + WT
   240 C1 = C1 + 1
   250  IF C1 <  = 5 THEN  GOTO 220
   260 AV = T1 / 5
*  270 IF AV < 180 THEN  GOTO 290
   275  PRINT "ACCEPT THIS LOT"
*  280  GOTO 295
   290  PRINT "REJECT THIS LOT"
   295  END
```

Program 2-5. Package-weight monitor.

We have included a REM statement describing the purpose of the program. Look at line 200. Note again that we have initialized a variable to zero even though we could let BASIC do it for us. Later, if we want to include this routine as part of a more complex program, the value of T1 will be reset to 0 every time this program segment is exercised. Failure to include such a statement would cause the value of T1 to grow ever larger as more and more lots are sampled. Thus the program would erroneously accept every sample after the first. (Doubtless, the computer would be blamed for this obvious programmer error.) Note that we have selected C1 for counting, T1 for totaling, WT for the package weight, and AV for the average. Selecting variable names carefully will make the meaning of each program statement clearer. Don't use A9 for weight or TF for counting. While TO might have been nice for the total, that 2-letter word is reserved. If we had used TO as a variable in line 200, Applesoft would have reported a ?SYNTAX ERROR. Soon we will discover what TO is reserved for. Line 270 determines which message will be displayed according to the average weight. Line 280 assures that we get exactly one message. Let's run the program.

```
]RUN
WT 1?182
WT 2?190
WT 3?180
WT 4?179
WT 5?177
ACCEPT THIS LOT

]
```

Figure 2-3. Execution of Program 2-5.

To check another lot, simply run the program again.

If this is all that the program does, it might be more practical to use a hand-held calculator. In practice, there are many more factors to consider in the above problem. While it may be illegal for packages to be underweight, it is unprofitable to sell overweight packages. So, in addition to checking for minimum average, our program ought to check for any package over a certain weight, say 185. Furthermore, there may be a legal minimum weight, say 178.

The program can also easily be modified to process several batches of data. Simply change

```
280  GOTO 295
```

to

```
280  GOTO 200
```

and replace

```
295  END
```

with

```
295  GOTO 200.
```

Now, how do we terminate execution of the program? We may enter a value of 0 to indicate that there are no more batches to process. Then the statement

```
232  IF WT = 0 THEN  GOTO 999
```

may be used to divert program execution to statement 999 for a weight of 0. We had better include the statement

```
999  END
```

to avoid the following error message:

```
?UNDEF'D STATEMENT ERROR IN 232
```

in Applesoft, or the message

```
*** BAD BRANCH ERR
STOPPED AT 232
```

in Integer BASIC. Special data values used as signals to control the action of a program are sometimes called *dummy data.* These changes are left as exercises.

While we could use CTRL-C in response to an INPUT request, it is not desirable to depend on this method. CTRL-C is more for programmers to use during program development. Our programs should provide for more orderly control. We often want further computing after the last INPUT

item. CTRL-C terminates the program, but the use of a special data value allows us to direct the program to continue processing.

.... SUMMARY

Know your problem well before coding your solution program. Have a plan. It is easier to make major changes on paper than it is to make major changes in a program that has already been typed into the computer.

The IF . . . THEN statement may be used to determine the next statement to be executed while the program is running.

Use REMarks freely, but properly. Don't state the obvious. State the purpose of a statement or group of statements. It is vital that REMs be accurate. Make sure that your program documentation keeps up with any changes in your program at all times. It is very frustrating to sort through program code that does not agree with the documentation. This can be worse than no documentation at all. However, don't use that comment as an excuse to omit REMs.

Artificial values (dummy data) may be used as data to control what statements will be executed next.

Problems for Section 2-1 .....

At this point, we know enough about BASIC to program solutions to a wide variety of problems. We could find the sum of the counting numbers from 1 to 1ØØ, or from A to B as long as we don't exceed 32767 in Integer BASIC or 1E38 in Applesoft. We could find sums of even integers or odd integers or those divisible by 5, etc. We could do something as simple as having the Apple display "I LIKE BASIC" some specified number of times. Use your imagination. You needn't limit yourself to the problems listed here. If you have an Apple to yourself, then you can answer all of those "I wonder what would happen if . . ." questions with, "I'll try it." The computer never raises its voice or remembers our "dumb" questions; it just beeps and tells us what is wrong.

AI 1. In the birthday problem (Program 2-4), have the computer print the amount received for this birthday and the total so far for each birthday.

A 2. In the package-inspection problem (Program 2-5), make the changes necessary to repeat processing for many batches of data in a single execution. Insert at least one blank PRINT statement so that the batches are separated on the screen.

AI 3. Rewrite the package-inspection program (Program 2-5) to test for a minimum package weight of 178 g and a maximum package weight of 183. Have the program report the reason for rejecting a lot and repeat for another batch.

A 4. Four test scores were 1ØØ, 86, 71, and 92. What was the average?

A 5. Write a program to count the number of odd integers from 5 to 1191 inclusive.

A 6. Write a program to find the number of and the sum of all integers greater than 1ØØØ and less than 2213 that are divisible by 11. (Start with 1ØØ1.)

A 7. Three pairs of numbers follow in which the 1st is the base and the 2nd is the altitude of a triangle: 1Ø,21Ø 12.5,8; 289,114. Write a program to print the area for each triangle. Use dummy data.

A 8. A person is paid $.Ø1 the 1st day, $.Ø2 the 2nd day, $.Ø4 the 3rd day, and so on, doubling each day on the job for 3Ø days. Write a program to calculate the wages for the 3Øth day and the total for the 3Ø days.

A 9. Write a program to print the integers from 1 to 15, paired with their reciprocals.

A 1Ø. A customer put in an order for 4 books that retail at $1Ø.95 and carry a 25% discount, 3 records at $4.98 with a 15% discount, and 1 record player for $59.95 on which there is no discount. In addition, there is a 2% discount allowed on the total order for prompt payment. Write a program to compute the amount of the order.

AI* 11. In the song "The Twelve Days of Christmas," gifts are bestowed upon the singer in the following pattern: the 1st day she received a partridge in a pear tree; the 2nd day 2 turtledoves and a partridge in a pear tree; the 3rd day 3 French hens, 2 turtledoves, and a partridge in a pear tree. This continues for 12 days. On the 12th day she received 12 + 11 + . . . + 2 + 1 gifts. How many gifts did she receive altogether? Another way to ask this question is to ask: If she had to return 1 gift each day after the 1st, on what day would she return the last gift?

AI* 12. For Problem 11, have the computer print the number of gifts on each of the 12 days and the total up to that day.

A 13. George took tests in 2 courses. For the 1st course the scores were 83, 91, 97, 1ØØ, and 89. For the 2nd course the scores were 65, 72, 81, and 92. Write a program that will compute both test averages. You will need 2 dummy-data values. One value will signal the end of this set of scores, and the other will signal the end of this execution of the program.

2-2...Random Events

How do they get the computer to flip coins, deal cards, or roll dice? These things are really very easy to simulate. All we need is the ability to gener-

ate numbers at random. BASIC includes exactly what we need for this. RND(X) produces a random number. RND is a little "black box" in BASIC that brings forth a number at random each time it is mentioned. Applesoft provides decimal numbers in the range Ø to .999999999; Integer BASIC provides integers. The number enclosed in the parentheses is called the *argument* and is very important.

Here are the rules for Applesoft:

RND(I)	I>Ø	Yields a different random value for each successive access.
RND(I)	I=Ø	Produces the last random number used.
RND(I)	I<Ø	Produces the same value each time the same value of I is used.

If we want the same sequence every time a program is run, then we must use a negative value for I for the first access and a positive number for all succeeding accesses. Then, to change the random sequence, simply use a different negative number or any positive number for the first access. The ability to repeat a random sequence is useful for program testing.

Here are the rules for Integer BASIC:

RND(I)	I>Ø	Yields an integer at random in the range Ø to I−1.
RND(I)	I=Ø	Produces the *** >32767 ERR error message. Avoid this value.
RND(I)	I<Ø	Produces an integer at random in the range Ø to I+1. Thus all nonzero values will be negative.

Let's look at an Applesoft program to print 10 random numbers.

```
100  REM * GENERATE A FEW RANDOM NUMBERS
200 I = 1
230  PRINT RND(1)
240 I = I + 1
250  IF I <  = 10 THEN  GOTO 230
9999  END
```

Program 2-6. Generate ten random numbers.

We have built a little counting routine that enables us to print RND(1) 10 times. Here is a sample RUN of our program.

```
]RUN
.936928502
.620447973
.98411756
.228080195
.0444301156
.929139704
.228214184
.954965809
.641032259
.946861798
```

Figure 2-4. Execution of Program 2-6.

Now we will adapt this new ability to flip a coin. Let's flip it 39 times to just fill 1 line of the screen without moving to the next line. There are 3 parts to this problem. We need to count to 39, generate a random flip, and print an H or a T depending. We know all about counting. We can decide whether to print an H or a T if we know how to tell which came up. All that remains is to organize how to distinguish heads from tails. We want half of each. So if we designate all of the random numbers from 0 to .499999999 as heads and all of the numbers from .5 to .999999999 as tails, the problem is solved in Applesoft. We merely test RND(1) in an IF . . . THEN statement. If we get less than .5, then branch to a statement that displays an H; otherwise "drop through" to a statement that displays a T. Following the PRINT "T"; statement, we must be sure to put in a GOTO statement to divert execution around the PRINT "H"; statement. Here is a program to do just that.

```
198  REM * FLIP A COIN 39 TIMES
200 FL = 1
230  IF  RND (1) < .5 THEN  GOTO 270
250  PRINT "T";
260  GOTO 280
270  PRINT "H";
280 FL = FL + 1
290  IF FL <  = 39 THEN  GOTO 230
999  END
```

Program 2-7. Flip a coin 39 times.

```
]RUN
THHHTTHTHTTHHTTHTHHHTHTTTHTHHHHTTTTHTHT
]
```

Figure 2-5. Execution of Program 2-7.

There you have it.

Using Integer BASIC, we simply replace statement 230 with

```
230 IF RND(2) < 1 THEN GOTO 270
```

We have accomplished what we set out to do. However, we really want to know as much about BASIC as possible. So, let's probe further.

....A RaNDom Exploration

The random-number generator may be bent to our needs in many ways. We have chosen to select 2 equal halves by forming a boundary at .5. This works fine for flipping a coin, but suppose we want to roll a die. Now there are 5 boundaries. We get numbers like .166666667 and .833333333. There is a much better way. If we multiply all numbers in the range of Ø to 1 (including Ø and excluding 1) by 6 then we get results in the range from Ø to 6 (including Ø and excluding 6). Then we could successively test to see if the result is less than 1, then less than 2, through 6, to get a value for the face of a die. This will certainly work, but it is not recommended. Once again Applesoft comes to the rescue. This time it is INT(N) that makes life simple.

....INT(N)

INT(N) is special for developing an integer value that is the greatest integer less than or equal to the argument. Thus, INT(3.9876919)=3, INT(4)=4, and INT(−9.8)=−1Ø. So, if we simply generate random numbers in the range from Ø to 5.99999999, then we can apply INT(N) to get integers in the range from Ø to 5. We merely add 1 to the values Ø to 5 to get values in the range 1 to 6. This is, of course, exactly what we want for rolling dice. Bingo—another problem solved. Let's look at Program 2-8 to roll a die 10 times.

```
   198  REM * ROLL A DIE TEN TIMES
   200 I = 1
   210 V1 =  RND (1) * 6 + 1
   220  PRINT V1, INT (V1)
   230 I = I + 1
*  240  IF I <  = 10 THEN  GOTO 210
```

Program 2-8. Roll a die ten times.

```
]RUN
2.28763901       2
4.31886087       4
2.39532511       2
6.4051106        6
3.13972997       3
1.87393362       1
6.51299232       6
1.10674357       1
4.19665986       4
5.15195719       5

]
```

Figure 2-6. Execution of Program 2-8.

To accomplish the same result in Integer BASIC, simply use RND(6) + 1 in place of RND (1) * 6 + 1.

.... IF . . . THEN Revisited

IF . . . THEN is used so frequently in BASIC to transfer program control that an abbreviated form exists.

```
240  IF I <  = 10 THEN 210
```

may be used in place of line 240 in our die-rolling program above. Using this new form of the IF . . . THEN statement, our coin flipping of Program 2-7 may be rewritten as follows:

```
   198  REM * FLIP A COIN 39 TIMES
   200 FL = 1
*  230  IF  RND (1) < .5 THEN 270
   250  PRINT "T";
   260  GOTO 280
   270  PRINT "H";
   280 FL = FL + 1
*  290  IF FL <  = 39 THEN 230
   999  END
```

Program 2-9. Program 2-7 showing shortened IF . . . THEN.

Lines 230 and 290 in Program 2-9 use the shortened form of IF . . . THEN.

.... SUMMARY

RND(X) provides a source of random numbers. In Applesoft, we get numbers in the range 0 to .999999999. RND(X) in Integer BASIC returns an integer from 0 to X−1 for positive values of X and from X+1 to 0 for negative values of X.

INT(X) in Applesoft returns the greatest integer not greater than X.

IF . . . THEN has an abbreviated form, which we may use for conditional transfer. 100 IF X<5 THEN 230 will transfer control to line 230 if X<5 is true.

Problems for Section 2-2 .....

AI 1. Modify the coin-flipping program (Program 2-9) to repeat the 39 flips 5 times.

AI 2. Modify the coin-flipping program (Program 2-9) to count the number of times tails comes up in 39 flips.

AI 3. Write a program to flip a coin 1000 times. Count the number of tails. You might choose not to display Hs and Ts.

AI 4. Write a program to roll 2 dice 10 times.
AI 5. Write a program to provide math drill problems in addition. Request limits and the number of problems using INPUT. Display the number of right answers at the end.

2-3...A Better Way to Count (FOR and NEXT)

Having written numerous counting loops, we imagine that there is some more compact method for doing this. After all, just about everything we do seems to involve counting of some sort.

.... BASIC Loops

FOR and NEXT in BASIC automate the control functions of a program loop. Thus our earlier program to count from 1 to 7 becomes Program 2-10.

```
   100  REM * COUNTING WITH FOR...NEXT
*  110  FOR C1 = 1 TO 7
   115  PRINT C1
*  130  NEXT C1
   999  END
```

Program 2-10. Program 2-3 using FOR . . . NEXT.

Statement 11Ø automatically establishes the limits on C1 as 1 and 7. Statement 13Ø automatically adds 1 to the value of C1 and tests to determine if C1 is less than or equal to 7. The value of C1 will be 8 when execution reaches line 999 of this program. Look again at line 11Ø. Now we know why TO is a BASIC keyword and must not be used in a variable name. If you want to save the last used value of the loop variable, then you need a statement such as 12Ø C2 = C1 in this program. It is important to note that the statements between FOR and NEXT will always be executed at least once. If we program the statement FOR X = 4 TO 1, then the loop will be executed for X = 4. The NEXT statement will add 1 to 4, getting 5, and then find that X is greater than 1, and execution will "drop through," behaving in exactly the same way as our "hand-built" loops. To count from A to B by 2s, simply code FOR C1 = A TO B STEP 2. We may STEP by −3 or even N. Applesoft allows decimal values, while Integer BASIC is, of course, limited to integers.

The FOR and NEXT statements provide several important benefits. FOR and NEXT loops execute faster than the identical hand-built variety. Their use reduces the number of ideas that we have to store in our heads as we write our programs. Those simple BASIC keywords embody the more complex controls actually used to construct the loop itself without requiring us to think about the detail each time that we use them, thus

freeing our mental processes for solving the specific problem at hand. The ability to make a small number of program statements represent complex solutions greatly simplifies the writing of correct computer programs.

Now we can think about some of the counting loops we have looked at before. Consider the birthday-dollars program (Program 2-4): in the original program, we had a line 11Ø C1 = 1. That line happened to be the opening statement of a counting loop, but that statement could set the value of C1 to 1 for zillions of reasons. On the other hand, the statement

```
11Ø  FOR C1 = 1 TO 21
```

is crystal clear. It can mean only one thing: we are going to do something 21 times. In exactly the same manner, NEXT C1 conveys much more information to the person reading the program than

```
15Ø  IF C1 < = 21 THEN  GOTO 12Ø.
```

FOR and NEXT are designed to go together. Don't try to initialize a loop with

```
1ØØ  C1 = 1
```

and later close it with

```
2ØØ  NEXT C1
```

Luckily, you will get the following message from Applesoft:

```
?NEXT WITHOUT FOR ERROR IN 2ØØ
```

and from Integer BASIC you will get

```
*** BAD NEXT ERR
STOPPED AT 2ØØ
```

Occasionally, you will be sure you have a loop to repeat something several times. But, alas, it only happens once, and the computer sends no error messages. While the computer requires that a NEXT statement be preceded by a FOR statement, it does not necessarily report that a FOR statement was not followed by a NEXT statement. Now you know.

.... SUMMARY

FOR and NEXT are paired up to control program loops in BASIC. For A = B TO C STEP D opens a loop by assigning the value of B to A. Each iteration of the loop is accomplished by adding the value of D to the value of A. When the value of A "goes past" the value of C, the loop is done. NEXT A causes the next iteration of the loop that was opened with the FOR A . . . statement. If STEP is omitted, the step value is assumed to be 1.

Problems for Section 2-3

For each of the problems here, use FOR and NEXT where appropriate.

AI 1. Modify the package-inspection program (Program 2-5) to use FOR and NEXT.

AI 2. Write a program to count the number of odd integers from 5 to 1191 inclusive.

A 3. Write a program to find the number of and the sum of all integers greater than 1000 and less than 2213 that are divisible by 11. (Start with 1001.)

A 4. A person is paid $.01 the first day, $.02 the second day, $.04 the third day, and so on, the amount doubling each day on the job for 30 days. Write a program to calculate the wages for the 30th day and the total for the 30 days.

A 5. Write a program to print the integers from 1 to 15, paired with their reciprocals.

AI 6. Do the "Twelve Days of Christmas" problem using FOR and NEXT.

AI 7. For Problem 6, have the computer print the number of gifts on each of the 12 days and the total up to that day.

AI 8. Modify the coin-flipping program (Program 2-9) to repeat the 39 flips 5 times.

AI 9. Modify the coin-flipping program (Program 2-9) to count the number of times tails comes up in 39 flips.

AI 10. Write a program to flip a coin 1000 times. Count the number of tails. You might choose not to display Hs and Ts.

AI 11. Write a program to roll 2 dice 10 times.

AI 12. Write a program to provide math drill problems in addition. Request limits and the number of problems using INPUT. Display the number of right answers at the end.

A* 13. Examine the following program:

```
100  FOR I = 1 TO 1.3 STEP .1
110  PRINT I
120  NEXT I
```

What values do you think it will display? Run it. Do you get what you expect? Write a program to display the four values you expected.

PROGRAMMER'S CORNER 2

Screen Editing

We have been making changes in program lines by simply retyping the entire line. If the line we wish to change is a long one and we merely want

to change a character or two, retyping the whole line may be counter-productive because we may just end up making another typing error. Applesoft and Integer BASIC include a set of commands that allow us to move the cursor around the screen to change what is displayed there. Whatever appears on the screen is stored in memory. In fact, program lines that appear on the screen are stored in two places. One place is invisible to us. That is where BASIC keeps the entire program. When we type a line of a program, BASIC incorporates it into any existing program already stored in that invisible part of memory. The visible line on the screen is stored in the visible part of memory used for text display and for low-resolution (Lo-Res) graphics.

The best possible way to learn about these screen editing features for the Apple is to sit down with the computer and experiment. Try everything described here. Soon you will be doing all of this automatically.

....Arrow Keys

The two keys at the very right of the second row of keys from the bottom are marked with arrows. They can be used to save us a lot of typing effort. The left arrow key has the ability to erase a character from invisible memory while it appears on the display screen. The right arrow key has the ability to read a character from the display screen so that it may become part of the line that we are typing. Each time we press the left arrow key, one character is removed from the line. Every time we press the right arrow key, a character is read in from the display screen. Thus, if we have begun typing the line

```
100 PRINT "THIS IS A PRONT STAT
```

and we spot that we have misspelled PRINT we may immediately press the left arrow key eight times until the cursor is blinking directly on top of the incorrect letter O. Next, we type the letter I and press the right arrow key seven times until the cursor is again at the end of the line. Following this, we may finish the line as though we had never made a mistake. It is important to realize that we must retrace with the right arrow any characters removed with the left arrow. The line will be entered up to the cursor only. This technique may be used on any line that we are typing up to the time we press the RETURN key.

Every time the right arrow key is pressed a character is read in from the display screen. This is true even at the right end of the screen. The cursor will disappear from the right and appear suddenly at the left of the screen on the next line down. If we next press the left arrow, the cursor will disappear from the left of the screen and appear suddenly at the right of the screen up one line. If we press the left arrow when the cursor is at the left of the screen, we cannot pass over the BASIC prompt. The cursor will simply move down one line in the same horizontal position. Pressing one of the arrow keys and then pressing the REPT key will make the cursor

move repeatedly until the REPT key is released. All of this will be clear after a little experimentation.

Now suppose we have a slightly different situation. We have begun to type a line as follows:

```
100 PRINT "THISS IS THE BEGINNING
```

and we notice that we have the double "S". We can press the left arrow key and the REPT key until the cursor is blinking over the "S" we want to eliminate. Next strike the ESC key and then the "A" key. After this, press the right arrow key and the REPT key until the cursor is in position to add the closing quote. Add the closing quote and press the RETURN key. The double-S problem will be solved. The ESC A sequence is one of a set of controls that enable us to move the cursor around the screen without reading characters in or erasing them. The characters passed over in this way are ignored.

.... Cursor Controls A, B, C, and D

Two things happen when we press one of the arrow keys. The cursor moves one character position on the screen and one character is either "read in" or "read out." When we strike the ESC key and then strike the "A" key, only one thing happens. The cursor moves one character position. Nothing is read in or out. In the double-S example, we used this to move the cursor over the position occupied by a character which we wanted to eliminate. Striking ESC followed by B, C, or D results in the cursor moving left, down, or up respectively. In each case, only the cursor is moved. No characters are read in or out.

ESC	A	right
ESC	B	left
ESC	C	down
ESC	D	up

Table 2-1. A, B, C, and D cursor movers.

To move the cursor two positions to the right it is necessary to strike ESC then A and ESC then A again. Thus, two keystrokes are required for each position we wish to move the cursor.

These cursor movers make it possible to make changes in an existing program line without having to retype the whole line. Suppose we want line 3ØØ to become line 35Ø. We just get line 3ØØ on the screen with LIST 3ØØ. Then move the cursor to the beginning of the line with a sequence of cursor movers. Strike the right arrow once. Strike the "5" key to replace the first zero in 3ØØ. And press the right arrow key and the REPT key until the cursor traces out the remainder of the line. Strike the RETURN key, remove line 3ØØ by typing 3ØØ RETURN, and the job is done.

The cursor movers ESC A, ESC B, ESC C, and ESC D are available in both BASICs.

.... Cursor Control in ROM Applesoft

ROM Applesoft includes a more convenient set of cursor controls in I, J, M, and K. To move the cursor, simply strike the ESC key followed by any number of keystrokes from the I, J, M, and K to move the cursor into position. I is up, J is left, M is down, and K is right. These cursor movers may be used in conjunction with the REPT key to locate the cursor quickly to a desired position.

ESC followed by:

I	up
J	left
M	down
K	right

Table 2-2. Cursor movers in ROM Applesoft.

With a little practice, you will be using these ESC sequences routinely to edit your program lines and commands.

.... POKEing for Easy Editing

Once we have tried to edit a line such as the following:

```
100  PRINT "THIS IS THE TIME TO T
     HINK ABOUT AN EASIER WAY TO
     EDIT"
```

we begin to wonder if there isn't some easy way to avoid skipping over the spaces inserted by Applesoft during the LISTing. We can use POKE to change the width of the display screen. The memory address for this is 33. Thus POKE 33,W sets the width. The value of W may be from 1 to 40. It turns out that if the width is 33 or less, those bothersome spaces in the program listing are eliminated. Simply type

```
POKE 33,33
```

and LIST again. This is what it looks like:

```
100  PRINT "THIS IS THE TIME TO T
HINK ABOUT AN EASIER WAY TO EDIT"
```

Now there are no crazy spaces to keep track of and skip over. Of course we need to set the width back with

```
POKE 33,40
```

For more about POKE see Appendix C.

Chapter 3

Apple Graphics (Lo-Res) and Much More

....A Graphic Example

It is one thing to program a computer to simulate the roll of a die and display a numeric result. It is another to program a computer to display a realistic picture of the die. One of the nice features of the Apple is its ability to produce color graphics. Let's use it. This will be surprisingly easy to do. Assume for the moment that we can assign colors, plot small blocks, and draw lines. Meanwhile, let's concentrate on the nature of a picture of one face of a die.

Think of drawing the six possible faces of a die on ordinary graph paper. This can be done nicely, if we use a rectangle five blocks wide and seven blocks high. We come up with the following sketch:

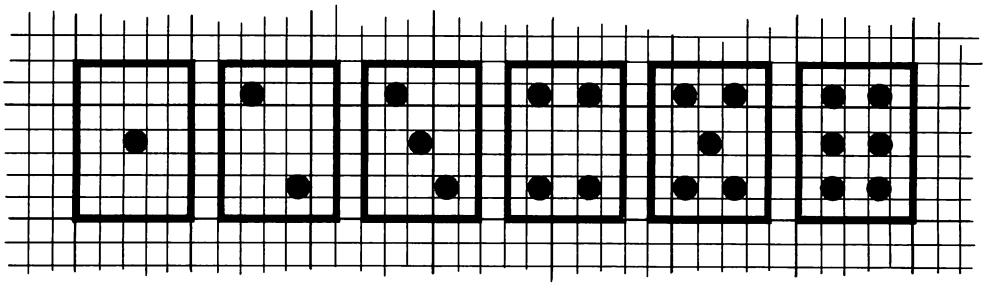


Figure 3-1. The six dice.

Now the computer problem separates into two parts. First, we need the die background. And second, we need six different configurations for the dots in some contrasting color. Let's see what solutions the Apple provides for these two problems.

3-1...Apple Graphics Keywords

There are six statements associated with graphics that we should know about before attempting to write a program. So, here we go.

.... The Graphics Screen

The statement

```
100  GR
```

prepares the Apple for graphics work. When this statement is executed, the screen is divided into 2 parts. The top part is now organized into 40 columns and 40 rows. Thus, we have 1600 blocks at our disposal. The remainder of the screen is reserved for 4 lines of regular text display.

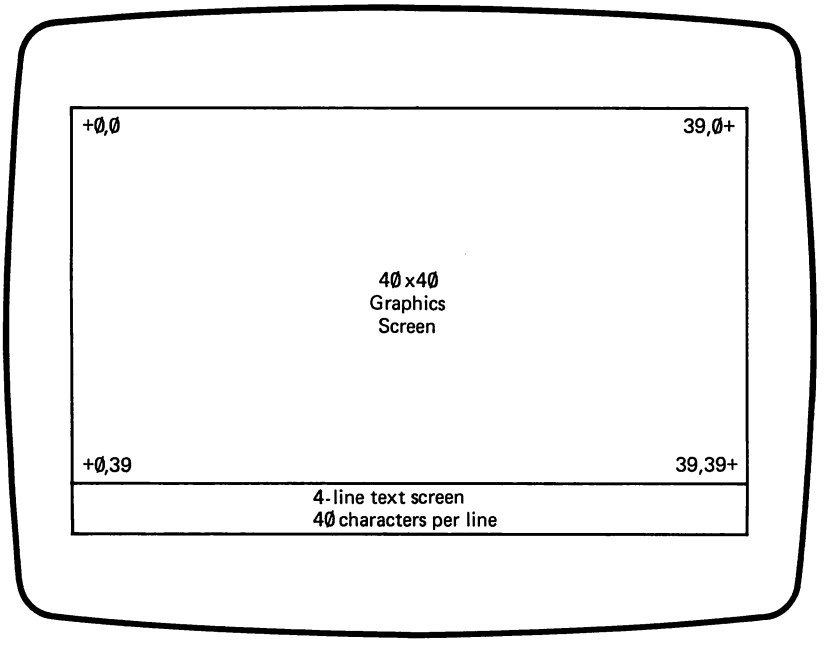


Figure 3-2. The graphics-screen layout.

This is called mixed graphics and text. Each block is identified by its column and row. The block in the upper left corner is labeled Ø,Ø. The block in the lower right corner is labeled 39,39. Columns are numbered from Ø to 39 from left to right, and rows are numbered from Ø to 39 from top to bottom. This is not the same as the conventional rectangular coordinate system widely used in mathematics, but this difference presents no great obstacle. The screen is not exactly square, so we call the plotted points blocks rather than squares. The Apple may be restored to the conventional text-oriented screen with the TEXT statement as follows:

```
19Ø  TEXT
```

....Apple Colors

Even if we are working with a noncolor monitor, we will have to pay attention to color. We will need to use at least white and black. There are 16 colors, numbered from Ø to 15 as follows:

Ø Black	8 Brown
1 Magenta	9 Orange
2 Dark blue	1Ø Grey
3 Purple	11 Pink
4 Dark green	12 Green
5 Grey	13 Yellow
6 Medium blue	14 Aqua
7 Light blue	15 White

Figure 3-3. Lo-Res Apple colors.

When the GR statement is executed, the Apple is set to black. We can change this to white with:

```
11Ø  COLOR= 15
```

The COLOR= statement may used to establish any of the 16 colors listed above. Of course we may use a statement like COLOR= C1 to assign the desired color. Nothing visible happens when a COLOR= statement is executed, just as nothing visible happens when a conventional assignment statement is executed. All plotting will appear in the most recently assigned color.

.... Plotting Points (Blocks)

Appropriately enough, we plot with the PLOT statement. The statement

```
5ØØ  PLOT 2,3
```

will plot a block near the upper left corner of the graphics screen in the color that is active when line 5ØØ is executed. Of course, we may use PLOT X,Y so that values may be calculated to establish a position before

executing the PLOT statement. Even

```
PLOT X + 3 * Y,2 * Y - 1
```

may be coded. It's that simple.

.... Drawing Lines

We could plot blocks next to each other with several PLOT statements to draw lines. However, BASIC includes special statements to draw horizontal and vertical lines for us.

```
600  HLIN 0,39 AT 0
```

will draw a horizontal line 40 blocks long at the very top of the screen.

```
700  VLIN A,B AT C
```

will draw a vertical line running from A to B in column C. We must assure that the values of A, B, and C remain within the 0 to 39 range to avoid peculiar results or even having our program terminate with an error message.

This is an area in which we can learn a great deal using immediate mode. We can issue the GR command and then set a color with COLOR=. Then it is a simple matter to PLOT points and draw lines directly from the keyboard. We can very quickly acquire a feel for the structure of the graphics screen and full-color Lo-Res graphics.

VLIN is very convenient for drawing bar graphs. We can incorporate some labeling in the four-line text screen at the bottom to make nicely readable charts.

.... Drawing a Die

The 5 BASIC keywords GR, COLOR=, PLOT, HLIN, and VLIN are all we need to do wondrous things with the graphics screen. We can now plan how to apply them to draw a die. Let's first draw the "1" face of a white die. We need to turn on graphics, set COLOR= to white, draw 5 vertical lines 7 blocks high, set COLOR= to black, and PLOT a block in the middle of the 5-by-7 rectangle. If we include the TEXT statement in our program, the graphics screen will disappear immediately after the program has run. So, let's omit that statement for the moment. Program 3-1 draws a "1" near the upper left corner of the screen:

```
98  REM  THE "1" FACE ON A DIE
100  GR
110  COLOR= 15
120  FOR I = 1 TO 5
130  VLIN 1,7 AT I
150  NEXT I
160  COLOR= 0
170  PLOT 3,4
180  END
```

Program 3-1. Draw the "1" face of a die.

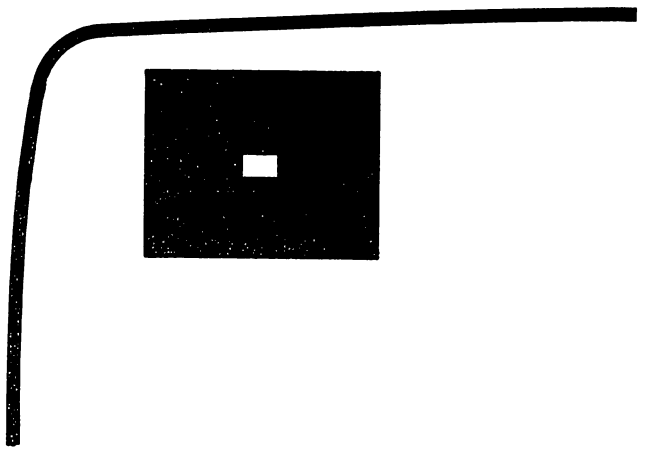


Figure 3-4. Execution of Program 3-1.

That is pretty nice. How do we get a "3"? Simply add the following 2 statements and run the new program:

```
165  PLOT 2,2
175  PLOT 4,6
```

After we have had a chance to study the graphics screen, we can type TEXT to clear it. TEXT may be used with a line number in a program. Or TEXT, like NEW, LIST, and RUN, may be used without a line number as a command. The 7 positions on the face of a die where a dot may appear are: 2,2; 2,4; 2,6; 3,4; 4,2; 4,4; and 4,6. By properly selecting from among these 7, we may draw any of the 6 faces.

.... SUMMARY

With 6 BASIC keywords, we control the Lo-Res graphics screen on an Apple. GR prepares the screen for us. We get an array of blocks laid out in 4Ø rows and 4Ø columns. In addition we retain a 4-line text screen at the bottom. We may set one of 16 colors in the range Ø to 15 with the COLOR= statement. We may plot points with the PLOT statement. Lines are easy to draw with the HLIN and VLIN statements. The rows and columns are numbered from Ø to 39 beginning in the upper left corner of the screen. Finally, we may erase the graphics screen and recover full use of the text screen with the TEXT statement.

Problems for Section 3-1 .....

AI 1. Write a program to display a die showing the "6" face in the upper right corner of the screen.

AI 2. Write a program to display a pair of dice, one showing a "1" and the other showing a "3".

AI 3. Write a program to request a number from 1 to 6 and display the appropriate die (try using colors if you have a color monitor).

AI 4. Write a program to draw a bar graph picturing the following dollar sales in thousands for a 9-week period:

Week	Sales
1	3Ø
2	27
3	26
4	31
5	26
6	3Ø
7	38
8	36
9	34

3-2...Divide and Conquer (Subroutines)

Once we have written the code to display a die of a particular color having a particular face value in a particular place, it is hard to be inspired to write new code to display that same die in another location or another color. And it is even less exciting to consider displaying five dice this way. When we find ourselves writing routine after routine, each of which is only a slight variation of the one just finished, programming becomes tedious. The more experience we gain in programming, the more opportunity we will have to utilize what we have already done. Often a current problem is only a slight variation of an old, already solved one.

If we want to display a green die and then a pink die in the same location, the only thing that changes is the color. Clearly, it is a nuisance to duplicate the code that does the actual graphing. We can easily isolate that code and direct the computer to execute it at will using GOSUB and RETURN.

.... GOSUB and RETURN

GOSUB 1ØØØ causes the computer to execute line 1ØØØ next regardless of the next statement numerically in sequence. However, GOSUB 1ØØØ differs from GOTO 1ØØØ in that GOSUB remembers its place in the program. When a RETURN statement is encountered, execution resumes following the most recent GOSUB. The program statements that begin with the line number following the keyword GOSUB and ending with a RETURN statement are grouped and referred to as a subroutine. Thus GOSUB means "GO do the SUBroutine."

For our green-die-followed-by-pink-die problem we need to have the program pause between the two displays. Otherwise, things will happen so quickly that we will not see the first die. This pause can be accomplished with a time-waster FOR . . . NEXT loop that does nothing else. The problem is solved in seven easy steps as follows:

1. Enable graphics mode.
2. Set green color.
3. Display the die.
4. Waste some time.
5. Set pink color.
6. Display the die.
7. End.

Putting off for the moment writing the actual die-display subroutine, let's look at a program to display our green and pink dice. See Program 3-2a.

```
   98  REM * CONTROL DIE DISPLAY
   100  GR
   110  COLOR= 12
*  120  GOSUB 1000
*  152  FOR X = 1 TO 1500
*  154  NEXT X
   160  COLOR= 11
*  170  GOSUB 1000
   190  END
```

Program 3-2a. The control segment of a die-drawing program.

We have been able to embody a group of statements in the single statement

```
120  GOSUB 1000
```

Again, we have a method for organizing our thoughts more easily by concentrating many computing steps in a single statement. We can think of

```
GOSUB 1000
```

as "display a die" without having to think about the actual BASIC statements required to do the display. Look at 152 and 154. Those 2 lines make up a delay loop. For a longer delay, use a value larger than 1500. Without a delay, we would not even see the 1st die because it would be so quickly replaced with the 2nd die.

And finally, the display routine is very easy. We may simply select those statements from our earlier die-drawing program and use appropriate line numbers. We may concentrate on the display without having to think about other parts of the program. We know that the 1st line should

be numbered 1000 and that the last statement should be RETURN. See Program 3-2b.

```
998  REM * DISPLAY A "1" DIE
1000  FOR I = 1 TO 5
1010  VLIN 1,7 AT I
1020  NEXT I
1030  COLOR= 0
1040  PLOT 3,4
1050  RETURN
```

Program 3-2b. Subroutine to display a "1" die.

Programs 3-2a and 3-2b together make up a complete program to display the "1" die in two different colors with a brief delay in between.

It is important to realize the impact of the END statement at 190 in Program 3-2a upon the subroutine beginning at line 1000. It is improper to execute a RETURN statement without a matching GOSUB. If we fail to obey this rule, Applesoft will deliver the following:

```
?RETURN WITHOUT GOSUB ERROR IN 1050
```

whereas Integer BASIC complains in the following way:

```
*** BAD RETURN ERR
STOPPED AT 1050
```

190 END assures that the routine at line 1000 is not executed an extra time.

Soon we will see that it is useful to separate the pieces of the program even fu_ther by using another subroutine to display the dots on the die. This will enable us to set a color for the dots easily and to plot any of the six possible faces.

.... Make It Handle the General Case

Wouldn't it be nice to be able to display a die anywhere on the screen? With the idea of subroutines well in hand, this new twist is easy. All we need is to "send" to our subroutine values that specify where a corner of the die is to be. We have already utilized this idea. We "sent" different color codes to the same subroutine at line 1000 in Program 3-2. Using X and Y as the horizontal and vertical position of the upper left corner of the die, we get the following subroutine to display the "1" anywhere on the screen.

```
998  REM  DISPLAY A "1" DIE
1000  FOR I9 = 0 TO 4
1010  VLIN Y,Y + 6 AT X + I9
1020  NEXT I9
1030  COLOR= 0
1040  PLOT X + 2,Y + 3
1050  RETURN
```

Program 3-3. Drawing a "1" anywhere on the screen.

However, we must assure that the values of X and Y place the entire die within the 40-by-40 graphics screen. That means that X may range from 0 to 35 and Y is limited to values from 0 to 33 for our 5-by-7 die face.

Now the final piece of the puzzle will fit into place as soon as we write 6 subroutines—1 for each of the 6 possible faces of a die. Numbering the first lines 1100, 1200, to 1600 will help to identify the purpose of each subroutine. Thus:

```
1098  REM  PLOT ONE
1100  PLOT X + 2,Y + 3
1110  RETURN
1198  REM  PLOT TWO
1200  PLOT X + 1,Y + 1
1210  PLOT X + 3,Y + 5
1220  RETURN
      .
      .
      .
1598  REM  PLOT SIX
1600  PLOT X + 1,Y + 1
1610  PLOT X + 1,Y + 3
1620  PLOT X + 1,Y + 5
1630  PLOT X + 3,Y + 1
1640  PLOT X + 3,Y + 3
1650  PLOT X + 3,Y + 5
1660  RETURN
```

Now we may remove lines 1030 and 1040 from our die-display subroutine. The display separates nicely into showing the background and plotting the spots. These 2 functions are now done with distinct subroutines. GOSUB 1000 displays the background. GOSUB 1100 through GOSUB 1600 may be used to display 1 through 6 spots on the die. We can set the colors independently. Once a die has been drawn on the screen, we can set the color to 0 and call upon the background display routine to erase the die, spots and all.

It is clear that once we have a number, such as R, that tells us how many dots to plot on a die, we need a way to branch to the appropriate subroutine. Thus, we wish to execute just one of the following statements:

```
GOSUB 1100
GOSUB 1200
GOSUB 1300
GOSUB 1400
GOSUB 1500
GOSUB 1600
```

We could do that with the following logic:

```
910  IF R < > 1 THEN 920
912  GOSUB 1100
914  GOTO 990
       .
       .
       .
950  IF R < > 5 THEN 960
952  GOSUB 1500
954  GOTO 990
960  IF R < > 6 THEN 990
962  GOSUB 1600
964  GOTO 990
990  RETURN
```

However, all that typing is cumbersome, and it requires 18 statements to perform a very simple decision. Our goal is always to simplify things. We could eliminate the 6 GOTO 990 statements, which are not essential. That would leave us with 12 statements, but we still have a choppy structure that is unnecessarily long and difficult to read (for humans—the computer doesn't care).

....Another Visit with IF . . . THEN

We can use a new feature of IF . . . THEN to simplify the decision as to which of the 6 die-display subroutines to execute. This new feature makes it possible to achieve the same result with 6 simple BASIC program lines.

Any BASIC statement may follow THEN in an IF . . . THEN statement. We may execute just one of the die-display subroutines with the following code:

```
910  IF R = 1 THEN  GOSUB 1100
920  IF R = 2 THEN  GOSUB 1200
930  IF R = 3 THEN  GOSUB 1300
940  IF R = 4 THEN  GOSUB 1400
950  IF R = 5 THEN  GOSUB 1500
960  IF R = 6 THEN  GOSUB 1600
```

Not only is this shorter to type, but it is much clearer to read. For any value of R in the range 1 to 6, just 1 of the IF . . . THEN tests comes out true. The other 5 come out false. Thus, the computer executes all 6 IF tests no matter what. But the computer is very fast, and the 5 false results will not delay execution noticeably for our present problem. Combining this feature with random numbers, we can program a wide variety of events.

Problems for Section 3-2 .....

AI 1. Write a program to display a die face showing a "6" in the upper right corner of the graphics screen.

AI 2. Write a program to display a random die face in the upper left corner of the screen.

AI 3. Display a random die face, leave it for a few seconds, and then erase it.

AI 4. Display two dice at random next to each other in the lower left corner.

AI 5. Write a program to display a blinking die. Let it blink 10 times, then leave the display on the screen.

AI 6. Display a few dice at random in random locations on the screen to simulate physically rolling the dice. Then display a pair of dice at random and leave them on the screen.

3-3...BASIC Multiple Features

.... GOSUB Revisited

At first we saw how we might use 18 statements to implement branching to 1 of 6 die-display subroutines. We reduced this to 6 easier-to-read lines using an extended feature of IF . . . THEN to execute any statement if the tested expression is true. Now we reduce this even further with a new feature of the GOSUB statement.

In Applesoft, we can accomplish the decision of the 6 IF statements with the multiple GOSUB capability.

```
910  ON R GOSUB 1100,1200,1300,1400,1500,1600
```

Should the value of R be less than 1 or greater than 6, the statement 910 will be ignored. However, values less than 0 or greater than 255 will be rewarded with Applesoft's

```
?ILLEGAL QUANTITY ERROR IN LINE 910
```

message.

In Integer BASIC, an alternative form of GOSUB is available.

```
892 REM * SELECT SPOT PLOTTING SUBROUTINE
894 REM  R = 1,2, ... ,6  YIELDS
896 REM  1100,1200, ... ,1600
898 REM  DISPLAY 1,2, ... ,6
910 GOSUB 100*R+1000
```

Integer BASIC allows variables and expressions as line-number references in GOTO, GOSUB, and IF . . . THEN statements. Therefore we

may construct a formula that produces the desired line number for any value of some variable. Along with this powerful and interesting feature, we issue an impassioned plea. Be careful how you use it! Plan the structure of your program with special diligence. Be sure to document clearly the logic of your formula. Changes in your original plan can introduce errors in logic that are very difficult to find. The line number 10Ø0*R+1ØØØ must exist for all values of R generated during program execution or you will get the message

```
*** BAD BRANCH ERR
STOPPED AT 91Ø
```

Now even the relatively simple six-statement logic used to branch to the proper spot-plotting subroutine has been reduced to a single statement. Lest we get the idea that all programs can be reduced by at least one statement (and therefore eliminated entirely), be assured that there is a limit to the features available in BASIC or any other computer language. Computers are finite and therefore limits do exist. Computers and computer languages are amazing, but they cannot perform magic.

.... Nested GOSUBs

We put subroutines to good use in the die drawing of the last section. It is worth noting that some plot statements were repeated. A three is just a one superimposed on a two. Thus:

```
13ØØ  PLOT X + 1,Y + 1
1310  PLOT X + 2,Y + 3
1320  PLOT X + 3,Y + 5
1330  RETURN
```

becomes:

```
13ØØ  GOSUB 12ØØ
1310  GOSUB 11ØØ
1320  RETURN
```

Four is two with two extra spots and can be plotted as follows:

```
14ØØ  GOSUB 12ØØ
1410  PLOT X + 3,Y + 1
1420  PLOT X + 1,Y + 5
1430  RETURN
```

Similarly, five is one superimposed on four, and six is a four with two extra spots.

We have "called" one subroutine from within another one. This is just fine and often very useful. Subroutines within subroutines are called "nested subroutines." We may nest subroutines up to 25 deep without incurring the wrath of Applesoft. Integer BASIC complains at 16.

What have we accomplished by all of this? There are two distinct benefits. We have made the spot-display subroutines more compact. Thus we can get more of the program on the screen at once. Once again, we have made the programming process more orderly through careful packaging. We have also included each of the spot-plotting statements exactly once. This reduces the possibility of error. If we have made an error, say the central spot is misplaced, we need look for a single PLOT statement to fix it for all die faces containing that spot. Any practice that makes programs easier to read, easier to change, or gives them better structure is to be encouraged.

.... GOTO Revisited

GOTO has the same multiple line-number branching capability as GOSUB. Thus the single Applesoft statement:

```
100  ON N1 GOTO 310,320,330,340,350,360,370
```

replaces seven IF . . . THEN statements. And the single Integer BASIC statement:

```
100 GOTO 10*N1+300
```

replaces a number of statements represented by the maximum value for N1. The same restrictions apply to the legitimate range for GOTO as for GOSUB.

Suppose we have a situation in which we want to execute 1000, if N1=3; 1100, if N1=6; and 1200, if N1=11. Do not be tempted to use the multiple GOTO or GOSUB capability of BASIC. In such a situation, it is much clearer to code three IF . . . THEN statements. Having 11 line numbers, only 3 of which are real, is very confusing to anyone reading your program. Don't do it! Even you won't understand it next week.

.... Multiple Statements

The ability to place several program statements on a single numbered line has some useful applications. Suppose we have a subroutine at 500 that requires that we set values for A, B, and C. This will generate several sets of lines of the following form:

```
100 A = 5
110 B = 9
120 C = 3
130  GOSUB 500
```

Where certain statements naturally belong together, it is nice to be able to

place them all on the same line. Using the colon (:) to separate statements, we may use the following equivalent code:

```
100 A = 5 : B = 9 : C = 3 : GOSUB 500
```

While it may be very nice to place several statements on the same line, there may be good reasons not to. It makes the line a little harder to edit. It may make the program harder to read when the statement LISTs on two or more lines. This capability should be used with caution.

Line numbers do require memory. Occasionally, a program grows to the point where it is too big for the available memory. One method for reducing the amount of memory a program requires is to use multiple statements per line. In doing this to an existing program you must be careful that you don't change the logic of the program by incorrectly combining lines that are referenced by a GOTO, an IF . . . THEN, or a GOSUB.

.... Multiple Statements and IF . . . THEN

BASIC allows multiple statements following IF . . . THEN. So a statement such as

```
100  IF A = 5 THEN B = 6 : C = 11
```

is perfectly legal in Integer BASIC and Applesoft. However, look out! That statement behaves radically differently in the 2 BASICs. In Applesoft, that statement will execute both B = 6 and C = 11 when A = 5, and neither B = 6 nor C = 11 when A does not equal 5. However, Integer BASIC will always execute C = 11 whether or not A = 5 and, it executes B = 6 only on the condition that A = 5. You are hereby warned.

It may be better to write code to implement the above logic as follows:

```
100  IF A <  > 5 THEN 120
110  B = 6 : C = 11
120  rest of the program
```

This is crystal clear and behaves in exactly the same way in both BASICs.

The fact that you know a certain feature does not mean that you should use it frequently or even at all. It is good to have a broad collection of capabilities available for use in the appropriate situation. It is also good to be aware of as many features as possible so that you can understand other people's programs. Know the language and use it well. It is a mistake to bend the logic of a program so that you can use some cute program statement. Cute or tricky programs are difficult to read. Some programmers like to embed tricky logic in their programs that "nobody will ever figure it out." That is just why you should not do it. Even you will never figure it out later when you want to change it.

PROGRAMMER'S CORNER 3

More Lo-Res Graphics .....

.... What Color is This?

With 16ØØ points to plot on the Lo-Res screen, how can we ever remember what color is where? We don't have to remember; BASIC does it for us.

```
SCRN(X,Y)
```

may be used to determine the color of the block at the point X,Y. The value returned by SCRN is in the range from Ø to 15. The values of X and Y may be in the range from Ø to 255. However, anything outside the range from Ø to 39 is off the normal graphics screen created by the GR command.

If we use SCRN(X,Y) in TEXT mode, we still get values in the range from Ø to 15, but these values don't correlate with colors. They correlate with characters on the text screen. It turns out that a character on the text screen occupies the same memory cells as 2 Lo-Res graphics blocks. So, to find the code used for a character displayed on the text screen, we could use a statement such as

```
PRINT SCRN(X,Y)+16*SCRN(X,Y+1)
```

where the value of Y is an even number.

.... Full-Screen Graphics with POKE

The normal Lo-Res graphics screen is established with the GR command. This leaves four lines of text screen at the bottom. We may then set up the screen to use those four text lines as eight graphics lines with

```
POKE -16302,Ø
```

When this is executed, the bottom of the screen will display some graphics lines and some blocks at the left. These lines and blocks can be cleared with just a few lines of code. If this is something we will be doing regularly, it will make sense to write a subroutine such as Program 3-4.

```
3992 REM  * SET FULL SCREEN GRAPHICS
     AND CLEAR BOTTOM OF SCREEN TO BL
     ACK
4ØØØ GR
4Ø1Ø POKE -16302,Ø
4Ø2Ø FOR L=4Ø TO 47
4Ø3Ø HLIN Ø,39 AT L
4Ø4Ø NEXT L
4Ø5Ø RETURN
```

Program 3-4. Subroutine to set full-screen graphics and clear last eight rows.

Any printing the program might do will now appear as a mosaic of colored blocks in the bottom eight lines of graphics display. Furthermore, anything we type at the keyboard in immediate or in response to an INPUT request will also appear as a mosaic of colored blocks. So, programs that use the full graphics screen may not use printing or request input with the INPUT statement. To take input from the keyboard without using the INPUT statement see Programmer's Corner 4.

Chapter 4

Miscellaneous Features and Techniques

.... Introduction

Certain calculations and other processes are required so frequently in programming that high-level languages like BASIC supply them in nice packages. Many of these packages are called *functions*. Some of them are called *operators*. And some are just plain features. These tools are a tremendous convenience in any computer language.

We have already used the INT function in some of our earlier programs in Chapter 2. Remember? INT(X) returns the greatest integer that is less than or equal to X. INT(5.699) = 5 and INT(−4.Ø91) = −5. When we are working with decimal numbers it is often useful to round off results. We will explore some other uses for INT in this chapter.

RND is a package that gives us access to random numbers in a program. We used RND to good advantage in Chapter 2. RND may be used to add interest and variety to games. This function is invaluable for writing simulation programs. We can write a program to model a real-life situation. By changing various factors in a proposed solution to a business problem, we can predict results without imposing poor judgment upon a frustrated public. We may confine our failures to unpublicized runs of a computer program.

These BASIC packages and numerous others will reveal themselves as extremely useful.

It takes several BASIC statements to determine whether a number is positive, negative, or zero:

```
890  REM  * DETERMINE +, Ø, OR -
900  IF X > Ø THEN S = 1
910  IF X = Ø THEN S = Ø
920  IF X < Ø THEN S =  - 1
930  RETURN
```

Once we have written such a subroutine, we should test it. Then, every time we need such a calculation in another program, we must type the entire subroutine. The SGN function does the same thing:

```
130 S =  SGN(X)
```

In just the same way we can determine the absolute value of a number in BASIC with the ABS function.

```
140 A =  ABS(X)
```

Not only are these functions useful in that they save us a lot of programming effort and typing time; they provide some meaning to the statement in which they appear. SGN(X) conveys the idea that we are interested in the sign of the number, while X = T : GOSUB 9ØØ fails to convey just why we are invoking the subroutine at line 9ØØ and that the result is returned in S. We will have to read the code beginning at line 9ØØ or put in REMs to understand the meaning.

The number in parentheses following the function name is called the *argument* of the function. This value is "passed" to the function, and the result is returned in the entire expression.

Just as BASIC includes LET, GOSUB, END, IF . . . THEN, and FOR . . . NEXT, it includes features such as INT, RND, SGN, and ABS as elements of the language. This means that the necessary programming has been done for us and incorporated into BASIC. There are many advantages to this. The programming has been tested for us. The features will generally execute much faster than if we write the same calculations in BASIC. This is especially true for trigonometric and logarithmic functions.

.... Prompted INPUT

Often we have been printing messages as labels for our INPUT requests. This is always a good idea. BASIC provides a convenient way to include the prompting message right in the INPUT statement.

In Applesoft the statement

```
100  INPUT "ENTER HERE?";T1
```

will produce exactly the same results as

```
100  PRINT "ENTER HERE"; : INPUT T1
```

Any message enclosed within quotes in an INPUT statement will be displayed exactly as typed. Note that when we use this option the question

mark we have come to expect with INPUT statements is not displayed. If we wish to have a question mark, then we must include it within quotes. We might want some prompting symbol other than the question mark anyhow.

In Integer BASIC the statement

```
10Ø INPUT "ENTER HERE",T1
```

will produce exactly the same results as

```
10Ø PRINT "ENTER HERE"; : INPUT T1
```

Unlike Applesoft, Integer BASIC still displays the question mark. We can't avoid it. Note that the delimiter is a comma in Integer BASIC, whereas in Applesoft it is a semicolon. However, when requesting several numeric values on the same line, we still use a comma to separate the several values being entered during program execution.

4-1.1...Applesoft Numeric Functions ABS, SGN, RND, SQR, and INT

For general programming, the most common functions are ABS, SGN, RND, SQR, and INT. Functions that come with the language are sometimes called built-in functions.

As discussed earlier, ABS(X) returns the absolute value of X, and SGN(X) returns -1, Ø, or $+1$ as the value of X is negative, zero, or positive. RND is the random-number generator. RND(X) returns random decimal numbers in the range from Ø to 1 including Ø and excluding 1. If X is negative, the number returned is the same for every occurrence of that negative value of X. If the value of X is zero, the most recently generated random number is returned. If the value of X is positive, a different value is returned for each successive use of the RND function.

SQR(X) returns the square root of X. We could also code $X^{.5}$ to represent "X to the one-half power," but SQR is convenient and executes faster. Of course the value of X must not be negative. A negative argument in the SQR function will incur the wrath of Applesoft. If we insist on coding a statement such as

```
10Ø  PRINT  SQR( - 4 )
```

we will be subjected to the following message:

```
?ILLEGAL QUANTITY ERROR IN 10Ø
```

Once we gain familiarity with how these functions work, as we are thinking about ways to solve computer problems, they will come to mind as they are needed.

Suppose we are interested in finding factors of integers. Right away INT should come to mind. We may program the computer to compare

INT(N/D) with N/D. If they are equal, then D divides into N without remainder and D is a factor of N. If INT(N/D) does not equal N/D, then D is not a factor of N. For example:

```
INT(69/5) = 13
```

while

```
69/5 = 13.8
```

Clearly 13 and 13.8 are not equal, so 5 is not a factor of 69. On the other hand:

```
INT(69/23) = 3
```

and

```
69/23 = 3
```

Twenty-three is a factor of 69 and so is 3.

To find the largest factor of 1946, all that we have to do is write a little program that tries all of the values from 1945 down to 2. The first one that is a factor is the largest factor. Display it and terminate the program. While we are at it, we might just as well make this a somewhat general program. Let's make our program request a value for testing. See Program 4-1.

```
   100  INPUT "FIND LARGEST FACTOR OF? ";N
   120  FOR D = N - 1 TO 2 STEP  - 1
*  140  IF N / D <  >  INT (N / D) THEN 180
   150  PRINT D : END
   180  NEXT D
   200  PRINT N;" IS PRIME"
```

Program 4-1. Find largest factor.

Note line 140. If we have a divisor that does not go without remainder, then we perform the next test. If not, then we have the largest factor. Display it and quit.

```
]RUN
FIND LARGEST FACTOR OF? 1946
973
```

Figure 4-1. Execution of Program 4-1.

There is something about this program that may not be obvious unless we witness the execution. The computer has to think for over 10 seconds before producing the answer for N = 1946. And it would delay for over 20 seconds for N = 1949. The smaller the first factor, the longer the delay. Surely we could find the largest factor of 1946 faster by hand. So can the computer.

Decimal division on a computer takes time. We could save one division for each value of N by assigning N/D to an intermediate variable:

```
135 Q = N / D
140  IF Q <  >  INT (Q) THEN 180
```

The time saving is about 10%. While this might be worth doing, we should also carefully examine the method we have chosen for solving this problem.

Take the case of 1946. The largest factor is 973, and the smallest factor is 2. We could simply test our factors beginning with 2. When we have found the smallest factor, the largest factor may be found by division. Thus we have gone from 973 trial values in the FOR . . . NEXT loop of Program 4-1 to a single trial for this particular value of N. We have also gone from 10 seconds to a small fraction of 1 second. That is an improvement worth working on. What if we enter 1949? This new method will require 1947 trial values of D and just over 20 seconds to execute. So, this method only helps for values of N that have factors. We should continue asking questions and making observations that may lead to an improved method that also works for prime integers.

Let's return to the observation that the largest factor of 1946 is 973 and the smallest is 2. How are the rest of the factors paired? See Figure 4-2.

2	973
7	278
14	139
139	14
278	7
973	2

Figure 4-2. Factor pairs of 1946.

There are six pairs of factors. Each pair appears twice. How can we determine when we have found all of the unique pairs of factors? For every factor less than or equal to the square root of a number, the other factor will be greater than or equal to the square root. Once we are convinced of that, the rest is easy. We need only test divisors up to the square root. Simply change

```
120  FOR D = N - 1 TO 2 STEP  - 1
```

to

```
120  FOR D = 2 TO  SQR(N)
```

and change

```
150  PRINT D : END
```

to

```
150  PRINT N / D : END
```

This change in strategy reduces the number of tests for N = 1949 from 1948 to 43. That is significant and worth incorporating into our program. We can also use the intermediate variable Q to store N/D. Thus:

```
150  PRINT N / D : END
```

becomes

```
150  PRINT Q : END
```

See lines 12Ø, 135, 14Ø, and 15Ø of Program 4-2.

```
   100  INPUT "FIND LARGEST FACTOR OF? ";N
*  120  FOR D = 2 TO  SQR (N)
*  135 Q = N / D
*  140  IF Q <  >  INT (Q) THEN 180
*  150  PRINT Q : END
   180  NEXT D
   200  PRINT N;" IS PRIME"
```

Program 4-2. Find largest factor using SQR (N).

....Rounding Decimal Results

Another use for INT comes up when we work with dollars and cents where calculations come out in fractional cents. We would like always to round figures off to the nearest cent for printing. Anything that is .5 cents or more is "rounded up," and anything less than .5 cents is "rounded down."

We can convert dollars and cents to cents by multiplying by 1ØØ. Then if we add .5 cents, all values from .Ø to .49 will become values in the range from .5 to .99, while all values in the range from .5Ø to .99 will become values in the range from 1.Ø to 1.49. If we next apply INT, all decimal portions that were less than .5 disappear, and all values that were .5 or more result in 1 cent being added. Then we get from cents back to dollars and cents by dividing by 1ØØ. Thus we can round values to the nearest cent with a statement such as

```
200 D1 = INT( D * 100 + .5 ) / 100
```

Then we can easily write a little test program to verify our solution for rounding values to the nearest cent (and incidentally for rounding any values to the nearest hundredth). See Program 4-3.

```
   100  REM  * DEMONSTRATE ROUNDING
*  140  PRINT "DATA","ROUNDED VALUE"
   150  READ D
   160  IF D =  - 9999 THEN  END
   200 D1 =  INT (D * 100 + .5) / 100
   210  PRINT D,D1
   220  GOTO 150
   900  DATA  3.09123, 4.94561
   910  DATA     2390, -1.5102
   920  DATA    .0009, -1.4861
   990  DATA  -9999
```

Program 4-3. Rounding to the nearest hundredth.

We have included the labeling of line 140 to give the display some meaning.

```
]RUN
DATA              ROUNDED VALUE
3.09123           3.09
4.94561           4.95
2390              2390
-1.5102           -1.51
9E-04             0
-1.4861           -1.49
```

Figure 4-3. Execution of Program 4-3.

Note that this also handles negative values correctly. It is always a good idea to verify that our programs work properly for a wide variety of values. Even though the current problem doesn't require a particular class of values, it is desirable to test the program for them anyway. It is much easier to put the finishing touches on a routine while we are familiar with the problem than to return to it months later when we discover that we really do want to handle those previously unwanted values.

.... Compound Interest

Suppose we have $100 in a savings account at 5.5% compounded daily. How much will that be at the end of 1 year? We can easily write a little program to calculate that. There is a formula that gives compound amounts very nicely.

$$A = P (1 + I)^N$$

where

A = Amount
P = Principal
I = Interest rate per interest period
N = Number of interest periods

The raised N indicates "to the power." This is done in Program 4-4.

```
100  REM  * CALCULATE COMPOUND INTEREST
200 P = 100
210 I = .055 / 365
220 N = 365
300 A = P * (1 + I) ^ N
310  PRINT A
```

Program 4-4. Compound interest by formula.

Note that "^" is used as the symbol for "to the power" on the Apple. This symbol is generated by the character "SHIFT-N" on the Apple keyboard.

```
]RUN
105.653643
```

Figure 4-4. Execution of Program 4-4.

Now, since we have enough trouble buying anything with a whole cent, let alone .3643 cents, we might as well round that value off to the nearest cent. We can do that easily by replacing line 310 with

```
310  PRINT  INT( A * 100 + .5 ) / 100
```

This program tells us what our amount will be at the end of the year. What the program doesn't tell us is what has happened to the buying power of our money due to inflation. It doesn't tell us of the federal, state, and even city income taxes we may have to pay on the interest. However, a savings account is still better than hiding the money in a mattress.

That compound-interest formula works just fine if we are going to put $100 in the bank and leave it there. But suppose we decide to put $20 into the account on the 1st of each month. For simplicity, let's consider that each month has 30 days and that the year has 360 days. Let's put $100 in the bank on January 1 and then put $20 in on the 1st of the month each month all year. We can handle this nicely with a FOR . . . NEXT loop going from 1 to 12. See Program 4-5.

```
100  REM  * ADD $20 EACH MONTH
200 P = 100
210 I = .055 / 360
220 N = 30
300  FOR M = 1 TO 12
310 P = P + 20
*  320 A = P * (1 + I) ^ N
*  330 P = A
340  NEXT M
350  PRINT "$100 PLUS $20 EACH MONTH $ ";
360  PRINT   INT (A * 100 + .5) / 100
```

Program 4-5. Compound interest with money added each month.

Note that the amount at the end of each month becomes the principal for the next month. See lines 32Ø and 33Ø of Program 4-5.

```
]RUN
$1ØØ PLUS $2Ø EACH MONTH $ 352.94
```

Figure 4-5. Execution of Program 4-5.

.... Programmer-Defined Functions (DEF FN)

Often it is convenient to define a function of our own and use it at various places in our program. BASIC DEFined FuNctions serve this purpose. We can set up a rounding function at the beginning of our program and then use it wherever we need that same calculation. Our rounding function may be defined as follows:

```
11Ø  DEF FNR(X) = INT( X * 1ØØ + .5 ) / 1ØØ
```

To invoke our new function we code a line such as

```
36Ø  PRINT  FN R(A)
```

BASIC "knows" that we want the value of A in line 36Ø to be used wherever X appears in the function definition on line 11Ø. The Xs in line 11Ø simply hold places where values will be inserted whenever an "FN R" is encountered in an expression. The value of X at the time that the function-definition statement is executed has no effect on the outcome of the program. The variable used in parentheses in the DEFining statement is called a dummy variable since no calculations ever use its value. The calculations are based on whatever replaces the dummy variable. We may code things such as

```
FN R(12345)      FN R(12 * .Ø98)      FN R( RND(4) * 1ØØØ )
```

Let's rewrite Program 4-3 to demonstrate rounding with a defined function. See Program 4-6.

```
   1ØØ  REM  * DEMONSTRATE DEFINED FUNCTION
*  11Ø  DEF  FN R(X) =  INT (X * 1ØØ + .5) / 1ØØ
   14Ø  PRINT "DATA","ROUNDED VALUE"
   15Ø  READ D
   16Ø  IF D =  - 9999 THEN  END
*  21Ø  PRINT D, FN R(D)
   22Ø  GOTO 15Ø
   9ØØ  DATA  3.Ø9123, 4.94561
   91Ø  DATA     2390, -1.51Ø2
   92Ø  DATA    .ØØØ9, -1.4861
   99Ø  DATA  -9999
```

Program 4-6. Rounding to the nearest hundredth.

In Program 4-6 we have defined the rounding function in line 11Ø and used it to display values rounded off to the nearest hundredth in line 21Ø.

```
]RUN
DATA              ROUNDED VALUE
3.ØØ9123          3.ØØ
4.94561           4.95
239Ø              239Ø
-1.51Ø2           -1.51
9E-Ø4             Ø
-1.4861           -1.49
```

Figure 4-6. Execution of Program 4-6.

Defined functions provide a way for us to put together packages of calculations in a convenient form. This is an ideal way to do conversions of all kinds. Programmer-defined functions are limited to 1 program statement, but that allows us a lot of leeway. We may define up to 26 functions in any 1 program in this way—FNA(X) through FNZ(X). Calculations and processes that cannot be done in a single program statement are best coded as subroutines and invoked with the GOSUB statement.

Converting from Fahrenheit to Celsius and vice versa is easy with two defined functions:

```
1ØØ  DEF FN C(X) = 5/9 * ( X - 32 )
11Ø  DEF FN F(X) = (9/5) * X + 32
```

Wherever we want Celsius from Fahrenheit, we simply code FN C(Fahrenheit temp) and wherever we want Fahrenheit from Celsius, we code FN F(Celsius temp). And if we want to round off the results, we include

```
12Ø  DEF FN R(X) = INT( X * 1ØØ + .5 ) / 1ØØ
```

Now to display the Celsius temperature rounded off to the nearest hundredth, code the following line:

```
21Ø  PRINT  FN R( FN C(T))
```

where T is the Fahrenheit temperature. We can even define one function in terms of another defined function. Thus

```
13Ø  DEF FN T(X) =  FN R( FN C(X))
```

will calculate the rounded value with any reference to FN T(X).

One convenient use of DEF is to define a random number in terms of the range desired. A function to return a random number in the range 1 to X follows:

```
1ØØ  DEF FN R(X) = INT( RND(1) * X + 1 )
```

.... SUMMARY

ABS, SGN, RND, SQR, and INT are commonly used built-in functions available in Applesoft. They have good mnemonic association. We may also build our own functions with DEF FN. These functions allow us to define any calculation that will fit in a single program statement. More complex packages may be created with subroutines.

Problems for Section 4-1.1 .....

1. Write a program to find all prime factors of an integer by rewriting the essence of Program 4-2 as a subroutine and calling it repeatedly. Eliminate duplicates.
2. Write a program to compare the effect of considering the banking year to have 360 days instead of the 365 on the real calendar. Use 5.5% and 12.5% on $100000.
3. Compare daily compounding with monthly compounding for $1000 at 5.5% and 12.5% for one year.
4. Compound interest may also be calculated without the formula given in this section. We may simply build a loop that adds the interest at the effective interest rate once for each period in the time that the money is on deposit. Write a program to calculate interest this way and compare your results with those in the programs of this section. Compare a 365-day year with a 360-day year.
5. Write a program to convert temperatures from Fahrenheit to Celsius. Request Fahrenheit temperatures from the keyboard. Be sure to have a way to stop. Zero may not be the best value for terminating this program execution.

4-1.2...Integer BASIC Numeric Functions and Techniques

....Integer BASIC Numeric Functions

Integer BASIC provides only SGN, ABS, and RND. We have seen all of these before. We review them here for convenience.

SGN(X) returns −1 if X is negative, 0 if X is zero, and +1 if X is positive; ABS(X) returns the absolute value of X; and RND(X) is our source of random numbers. For RND(X), the value of X must not be zero. For positive values of X, RND(X) produces integers in the range from 0 to 1 less than X. So, to obtain values in the range from 1 to X, simply code RND(X) + 1. For negative values of X, we get values in the range from 0 to 1 more than X.

.... Factors in Integer BASIC: A Technique

Finding factors in Integer BASIC is quite straightforward. We use the fact that arithmetic is limited to integers to do this. The result of 7/2 is 3. The result of 3*2 is 6. Six is not equal to 7, so 2 is not a factor of 7. If C is the result for A/B, then we determine whether C*B is equal to A. If it is, we have a factor. Now, to find the largest factor, all we have to do is put this test in a loop running from 1 less than the number to 2. See Program 4-7.

```
   150 PRINT "FINDING LARGEST FACTOR"
   160 PRINT
   200 INPUT "ENTER AN INTEGER ",N
   210 FOR D=N-1 TO 2 STEP -1
*  220 A=N/D
*  230 IF A*D<>N THEN 280
   240 PRINT D : END
   280 NEXT D
   290 PRINT N;" IS PRIME"
   300 END
```

Program 4-7. Finding largest factor.

We could even code line 220 and 230 as a single line:

```
230 IF N/D*D<>N THEN 280
```

Here we have developed a simple technique for handling a situation that requires a special function in Applesoft. We will often find that the integer limitation is not really a handicap.

```
>RUN
FINDING LARGEST FACTOR

ENTER AN INTEGER ?1991

181
```

Figure 4-7. Execution of Program 4-7.

If we watch while the computer produces the answer 181, we will notice that there is a considerable delay. For large prime numbers the delay will be even greater. Thinking about factors a bit will help us find a way to cut execution time dramatically. What are the factors of 12? That is easy: 2, 3, 4, and 6. How are they paired?

```
2          6
3          4
4          3
6          2
```

Figure 4-8. Factor pairs of 12.

Each factor pair in this example is duplicated. For a perfect square, there would be an odd pair. For every factor greater than or equal to the square root, the other factor is less than or equal to the square root. We can easily find the smallest factor by going from 2 to the square root with a new FOR statement:

```
210 FOR D=2 TO N-1
```

To find the largest factor, simply divide. We have already done the division in line 220 of Program 4-7. So, instead of displaying D in line 240, we want to display A. Applesoft has a square-root function, but Integer BASIC doesn't. Not to worry: we simply test D*D. If D*D is greater than N, then we have passed the square root and the search is complete. See Program 4-8.

```
   150 PRINT "FINDING LARGEST FACTOR"
   160 PRINT
   200 INPUT "ENTER AN INTEGER ",N
*  210 FOR D=2 TO N-1
*  215 IF D*D>N THEN 290
   220 A=N/D
   230 IF A*D<>N THEN 280
*  240 PRINT A : END
   280 NEXT D
   290 PRINT N;" IS PRIME"
   300 END
```

Program 4-8. Find largest factor without square-root function.

We have changed Program 4-7 by adding line 215 to exit the FOR . . . NEXT loop when the square root has been passed, modifying line 210 to cause the loop to begin with 2, and changing line 240 to display the correct factor.

```
>RUN
FINDING LARGEST FACTOR

ENTER AN INTEGER ?32759

1927
```

Figure 4-9. Execution of Program 4-8.

Of course, we have to see these programs execute to visualize the dramatic improvement in speed.

....SUMMARY

ABS(X) returns the absolute value of X. SGN(X) returns −1, 0, or +1 as X is negative, zero, or positive. The Integer BASIC random-number generator RND(X) returns a random integer in the range from 0 to X−1 for

positive values of X and in the range from Ø to X+1 for X negative. RND(Ø) will produce an error.

We developed a way to find factors using integer arithmetic to our advantage. We found a simple method of terminating a search for factors by exiting the loop when the trial denominator passed the square root by squaring it to see if the result was greater than the original number to be factored.

Problems for Section 4-1.2 .....

1. Rewrite Program 4-8 as a subroutine. Use this subroutine to find all prime factors. Eliminate duplicates.
2. Write a program to convert temperatures from Fahrenheit to Celsius. Request Fahrenheit temperatures from the keyboard. Be sure to have a way to stop.
3. Have the computer select a number at random in the range of 1 to 99. Don't display it. Offer the user the opportunity to guess the secret number. Have the computer tell the user whether this guess is high or low. Count the number of guesses it takes to get it right.

4-2.1...More Applesoft Goodies

There are lots of features that we could get by without. In fact, many versions of BASIC do not include some of the packages we will be opening in this section. However, these features do make life interesting and often easier.

.... HOME

HOME clears the display screen and places the cursor in the upper left-hand corner. This is very nice for keeping the screen uncluttered.

.... FRE

FRE(X) is a function that returns the amount of free memory in bytes. The value of X may be any legal Applesoft number. It is handy to use FRE(9) or FRE(8) because the 8 and 9 are right there on the keyboard with the left and right parentheses. A byte corresponds to a single character in memory. It takes 2 bytes to store an integer and 5 to store a decimal value. Applesoft keywords each require 1 byte. Arrays and strings require several bytes in addition to the space required for the data to be stored in them. If we are working with arrays and we want them to be as large as possible, this function will save a lot of trial and error. Be sure to run the program before determining the amount of free memory. Even then we

should allow 5Ø or 1ØØ bytes, because the program may use more memory during future executions with different data.

.... SPEED=

The display speed may be set to any value in the range from Ø to 255.

```
SPEED= Ø
```

will set the display to the slowest possible speed. With SPEED= Ø the display is around 6 characters per second, while at SPEED= 255 we get more like 1ØØØ characters per second—Apple's normal speed. This can be set at will anywhere in a program or in immediate mode.

.... CTRL-S

When the computer is concentrating on its display it is exceedingly fast. So fast, in fact, that we may miss what we want to see. We can slow things down with SPEED=, but that slows everything down. We can temporarily halt the display by pressing CTRL-S. This is convenient for locating a piece of a program that we want to look at with LIST and for halting an executing program while we write down some values during program development, but our finished programs should not rely upon this feature.

.... FLASH, INVERSE, and NORMAL

FLASH, INVERSE, and NORMAL determine the appearance of the computer output on the screen. These commands do not change what we type at the keyboard, only what the computer outputs to the screen. The commands are quite mnemonic. We may use a flashing or inverse display for emphasis.

.... SPC and TAB

SPC and TAB are functions that must appear in a PRINT statement.

```
231  PRINT TAB(X);"MESSAGE"
```

will display the "M" in MESSAGE in the Xth column of the current line. The first column is labeled 1. If X equals 41, then the rest of the display begins in the first column of the next line on the normal screen. The TAB function cannot move the cursor to the left on the current line.

SPC(X) in a PRINT statement causes X spaces to be displayed. If X takes the display past the end of the current line, SPC moves to the next line and continues counting. The range for X is Ø to 255.

.... HTAB and VTAB

HTAB and VTAB are Applesoft keywords that provide absolute cursor positioning.

```
100  HTAB X : VTAB Y
```

will place the cursor at position X,Y on the screen regardless of where it was before the above statement is executed. The next printing begins from this point. Y may be in the range from 1 to 24. X may be in the range from Ø to 255, but Ø is treated as 256.

.... POS

The POS(X) function may be used to determine where on the line the cursor lies. The argument of this function is a dummy and has no effect upon the function itself. The positions on the line are counted from Ø to 39.

.... PDL

The Apple permits up to 4 "paddles" and 3 game "buttons." The paddles are numbered Ø through 3. PDL(X) returns a value in the range from Ø to 255 according to the rotational position of the paddle dial. The dial contains a resistor that works like a light dimmer switch. The paddles make it possible to allow the user to control a program without using the keyboard. Most often the paddles seem to be used for games. They could just as well be used to respond to questions with numbered answers. We could read the value of the paddle and display the corresponding answer number on the screen. When the user has selected the desired answer, they notify the program by pressing the button. We read that the button has been pressed with PEEK.

The 3 buttons are located at memory addresses – 16287, – 16286, and –16285 for buttons Ø, 1, and 2. To determine whether button Ø has been pressed get the value of PEEK (– 16287). If that value is greater than 127 then the button has been pressed since the last time that address was PEEKed. Program 4-9 is a little routine to use paddle Ø to accept responses in the range from Ø to 9.

```
90  REM  * DEMONSTRATE USE OF PADDLE AND GAME BUTTON ZERO
100 X =  PDL (0)
120 Y = X /  INT (256 / 9)
140  HOME
160  PRINT Y
170  FOR I = 1 TO 500 : NEXT I
180 Z =  PEEK ( - 16287)
190  IF Z < 128 THEN 100
200  PRINT "YOUR ANSWER ";Y
990  END
```

Program 4-9. Use paddle to enter responses Ø to 9.

Note that line 17Ø seems to do nothing. It is there to minimize the flickering of the display of the current value derived from the position of the

paddle dial. The higher the limit on the loop, the steadier the display, but if we make the limit too high then the program will not respond to the rotation of the paddle. Somewhere between 25Ø and 5ØØ seems about right.

.... GET

Here is a way to take input from the keyboard without getting any text display on the screen. When we are working with full-screen graphics, this is a way to allow the user to communicate with the program.

```
25Ø   GET A$
```

looks for a single character from the keyboard. The program will "hang" on this statement until a character is entered. The RETURN key need not be pressed in this case. Any character entered prior to executing the GET statement will be read in and used. See Programmer's Corner 4 for how to control this.

4-2.2...Integer BASIC Goodies

.... MOD

Integer BASIC has a very nice operator. A MOD B returns the remainder after dividing A by B. We can perform modular arithmetic easily. Or this can be used to determine whether or not B is a factor of A. If A MOD B equals zero then B is a factor of A. Otherwise, it is not. You might try this in the factoring problem we did earlier.

.... PDL

See the discussion for PDL in Section 4-2.1.

.... FLASH, INVERSE, and NORMAL

Integer BASIC requires a POKE at address 5Ø to control the mode of screen display.

POKE 5Ø,127 puts the Apple in FLASH mode for display generated by computer output. Characters entered from the keyboard are displayed in the normal mode.

POKE 5Ø,63 puts the Apple in INVERSE mode for all display generated by computer output. Characters entered from the keyboard are displayed in the normal mode.

POKE 5Ø,255 puts the Apple in NORMAL display mode for all screen output.

.... CALL −936

CALL −936 clears the screen and places the cursor at the upper left-hand corner of the screen.

....TAB and VTAB

We may place the cursor anywhere on the screen with the absolute positioners TAB and VTAB.

```
200 TAB 7 : VTAB 6 : PRINT "DONE"
```

will place the cursor on line 6 in the 7th column for the word "DONE" to be printed by the PRINT statement following. The range for VTAB is 1 to 24 and the range for TAB is 1 to 255. If the value for TAB exceeds 40, the display is pushed onto succeeding lines as necessary.

4-3...Other Applesoft Functions

SIN(X), COS(X), and TAN(X) all return the trigonometric values we would expect. The value of X must use radian measure. The inverse function ATN(X) is also provided. ATN(X) returns radian values in the first and fourth quadrants.

From these trig functions all the others can be derived. It is up to the programmer to determine the correct quadrants where that is a problem.

EXP(X) and LOG(X) are also provided. EXP(X) raises e (2.71828183) to the Xth power, and LOG(X) finds the natural log of X.

4-4...Logical Operators in BASIC

....AND, OR, and NOT

Often in a program there are several conditions that may determine the next course of action. We might want to execute a subroutine if AV>95 and SC<70. We can do this with AND.

```
300  IF AV > 95 AND SC < 70 THEN  GOSUB 900
```

will do the job. AND is one of the three *logical operators* in BASIC. BASIC evaluates the expression "AV>95". If that expression is true, BASIC sets its value to one. If that expression is false, BASIC sets its value to zero. The same goes for "SC<70". We can even assign logical values to variables:

```
290 L1 = AV > 95 : L2 = SC < 70
295  IF L1 AND L2 THEN  GOSUB 900
```

This is equivalent to the single statement 300 above. In line 290 the value of L1 is set to 1 if AV>95 is true and 0 if AV>95 is false. Similarly L2 becomes 1 or 0. And finally, in line 295, L1 AND L2 becomes 1 or 0. We can even assign L1 AND L2 to another variable if that suits our purpose.

OR does just what you would expect.

```
400  IF L1 OR L2 THEN  PRINT "TRUE"
```

Line 400 prints "TRUE" if either L1 or L2 is true.

NOT simply reverses the logical state of an expression.

```
350  IF NOT (L1 OR L2) THEN  PRINT "TRUE"
```

prints "TRUE" only if L1 is zero and L2 is zero. When BASIC assigns values to logical expressions, it selects zero and one for false and true, but other values may be used. In fact the two logical states in BASIC are zero and not zero. Thus:

```
 6  OR  7   =  1

  6 AND 7   =  1

6 AND NOT 7 =  0
```

What about 7 and NOT 7? Well, 7 is "true," so NOT 7 is "false."

PROGRAMMER'S CORNER 4

Controlling the Keyboard.....

We have used the keyboard to interact with many of our programs in Integer BASIC and Applesoft. We have used INPUT often to request data from the keyboard. As long as we wait for the INPUT statement to actually execute, all is well; however, if there is a delay prior to an INPUT request and we press a key during that delay, the INPUT statement takes that key as the first character of our response. Thus, if we accidentally press a key during some delay, we run the risk of entering incorrect data. If we press the RETURN key in such a situation and the request is for string INPUT, then the INPUT statement may process a string of zero characters and proceed on that basis. Even expensive programs commercially available produce strange results through failure to recognize this as a potential problem.

To visualize the problem, type in Program 4-10 and run it.

```
 90 CALL -936
100 INPUT X
120 FOR I=1 TO X : NEXT I
*150 INPUT X
160 IF X>0 THEN 100
170 PRINT "DONE"
180 END
```

Program 4-10. Demonstrate premature keyboard entry.

When Program 4-1Ø displays a question mark, enter 15ØØ and press the RETURN key quickly twice. Then take your hands off the keyboard. That request from BASIC to ?REENTER or RETYPE LINE is generated by line 15Ø. The second RETURN was read by the INPUT at line 15Ø. Since we entered a RETURN with no numeric value, BASIC coughs. This is easily prevented. There is a memory location that resets the keyboard for a new character.

We can reset the keyboard with

```
14Ø POKE -16368,Ø
```

Enter this new line in Program 4-1Ø and note the difference in behavior. This time the INPUT statement waits for a response from the keyboard because statement 14Ø knocked out the spurious extra RETURN character entered in response to the INPUT at line 1ØØ.

GET in Applesoft can be managed in the same way. Placing a POKE −16368,Ø just before the GET statement will assure that no spurious data is taken from the keyboard. On the other hand, it may be important not to knock out a character entered during some noninteractive portion of a program. We may in fact want to "look" at the keyboard to "see" if any key has been pressed since some specific prior event.

.... Read the Keyboard with PEEK(−16384)

Memory location −16384 reads the keyboard. If we strike a key, the code associated with the key struck is stored at that address. If the value stored there is greater than 127 then a key has been struck. The character is not displayed on the text screen. In fact, INPUT reads the keyboard and displays the character and performs a lot of other testing for us. This testing includes verifying string or numeric input and out-of-range testing. We certainly would be ill advised to use PEEK(−16384) for all input. This is handy for situations where we are doing full-screen graphics and want to "look" at the keyboard to see if certain characters have been struck. The instructions for such a program should be very clear about the commands that may be entered in such a situation. This PEEK thing is good for only a single character at a time. If 5 characters are entered before we do the PEEK, only the last 1 entered will be there for us to "see." To read multiple characters we need to put this logic in a loop to control the number of characters to be processed.

If we use this procedure for reading the keyboard, then we should be certain also to clear the keyboard with POKE −16368,Ø. The −16384 address may be read over and over again. So, the next INPUT statement would pick up the last keystroke. POKE − 16368, Ø will prevent this confusing condition.

Chapter 5

Character Strings and String Functions

Most of our work has used numbers and calculations. However, we have printed messages and labels by enclosing them in quotation marks in PRINT statements. The ability to handle nonnumeric data is important in working with computers. Such data is referred to as *string data.* String data may contain any of the letters, digits, and special characters available on the computer. Thus, string data comes in character strings.

Strings may be used for a name-and-address mailing list, for instructions telling how to use a computer program, as labels to make the displayed results more understandable, or as part-identification labels in an inventory-control system. We might simply use strings to make a game program more conversational. We can ask the player's name and use it in later displayed messages. BASIC provides a variety of features that make the handling of string data very convenient. There are string variables, which enable us to store and manipulate character strings. Using string variables and string functions, we can manipulate individual characters and groups of characters. We can even print a string in reverse order just for fun.

5-1.1...Applesoft Strings

Applesoft provides string variables and a host of useful string manipulation functions. A string variable is distinguished from a numeric one by using a dollar sign ($) as the last character in the variable name.

We may work with string variables in many of the ways in which we work with numeric variables. For instance, any of the following statements may appear in a program:

```
100  LET A$ = "FIRST"
100  READ A$
100  INPUT A$
100  PRINT A$
```

String variables may contain from 0 to 255 characters at a time. In order to READ A$ we must provide a corresponding DATA statement. If we want to include a comma in the string, then we should enclose the string in quotation marks. Without the use of quotation marks, any comma is interpreted as the end of the current DATA item. For example, Program 5-1 READs string DATA and PRINTs it for us to see.

In this program we introduce a little technique in Applesoft for making programs more readable overall. It turns out that we may get an almost blank line by entering a line number followed by a colon. This may be used to make a clear visual break between different parts of a program. Beginning with this program, we will use this often. Integer BASIC does not provide the same nicety.

```
   100  READ A$
   120  PRINT A$
   130  GOTO 100
   495 :
*  500  DATA  GEORGE M. COHEN, ABE LINCOLN
   510  DATA  JOAN OF ARC
```

Program 5-1. READ . . . DATA with strings.

The comma in line 500 is interpreted by Applesoft as a data separator or delimiter. We could have provided the same data for this program by typing as follows:

```
]500 DATA GEORGE M. COHEN, ABE LINCOLN,
JOAN OF ARC
```

Here the screen has automatically pushed characters to the next line as we type. When we LIST this statement, Applesoft will arrange things a little differently.

```
]LIST 500

500  DATA  GEORGE M. COHEN, ABE L
     INCOLN, JOAN OF ARC
```

For short data items we could avoid having the computer rearrange things by placing each data item on a single line. Doing this will take up additional memory. However, we are writing very short programs that don't

require much memory. So, we won't worry about memory use until we are writing very long programs. The most readable form follows:

```
100  READ A$
120  PRINT A$
130  GOTO 100
495 :
500  DATA  GEORGE M. COHEN
510  DATA  ABE LINCOLN
520  DATA  JOAN OF ARC
```

Program 5-2. Program 5-1 with reformatted DATA.

It is always worth a little effort to make programs more readable. As we gain experience with programming, this comes automatically.

```
]RUN
GEORGE M. COHEN
ABE LINCOLN
JOAN OF ARC

?OUT OF DATA ERROR IN 100
]
```

Figure 5-1. Execution of Program 5-2.

That "OUT OF DATA" message is a little disturbing. Good programs will never produce that message! In some situations, programs that end with an error message will fail to perform as desired. We should always provide for an orderly program termination. In this case we may simply add an artificial string-data item to the data list. Such a data item is sometimes called *dummy data.* We will use this artificial data item as a signal to the program that all of the data have been read. After line 100 and before line 120 we compare A$ to the signal data. Using "STOP" as the terminating signal the final program looks like Program 5-3.

```
  100  READ A$
* 110  IF A$ = "STOP" THEN 900
  120  PRINT A$
  130  GOTO 100
  495 :
  500  DATA  GEORGE M. COHEN
  510  DATA  ABE LINCOLN
  520  DATA  JOAN OF ARC
* 599  DATA  STOP
  900  END
```

Program 5-3. Using dummy data to terminate program execution.

Now our little demonstration program terminates in an orderly way. Of course, the actual signal is arbitrary, just so we select some value that will not be a real DATA item and test for that value.

Applesoft permits us to compare strings for order in much the same way in which we compare numbers using IF . . . THEN. The sequence used is known as ASCII (American Standard Code for Information Interchange). For strictly alphabetical strings, this code will alphabetize in the conventional order. ASCII places the digits Ø through 9 ahead of the letters of the alphabet. We can easily write a short program to demonstrate order comparison.

```
   95  REM  * COMPARES STRINGS FOR ORDER
   100  PRINT
   110  PRINT "A$";
*  120  INPUT A$
   130  IF A$ = "STOP" THEN 240
   140  PRINT "B$";
*  150  INPUT B$
   160  IF A$ < B$ THEN 220
   170  IF A$ = B$ THEN 200
   175 :
   180  PRINT A$;" IS GREATER THAN ";B$
   190  GOTO 100
   195 :
   200  PRINT A$;" IS EQUAL TO ";B$
   210  GOTO 100
   215 :
   220  PRINT A$;" IS LESS THAN ";B$
   230  GOTO 100
   235 :
   240  END
```

Program 5-4. String comparison in Applesoft.

Lines 12Ø and 14Ø are string INPUT requests. We have the same option to include a message in quotes right in the INPUT statement itself for strings that we have for numeric input. Lines 11Ø and 12Ø may be replaced with the following single statement:

```
110 INPUT "A$?";A$
```

As with prompted INPUT requesting numeric data, no question mark is displayed. To exactly match the two statements, 11Ø and 12Ø, we have included our own question mark in quotes.

This quoted-message thing is nice, but if we have a situation where we want to use the same INPUT statement to ask different questions, we will still have to use a PRINT statement that displays a message stored in a string variable.

```
]RUN

A$?WHAT'S THIS
B$?WHAT'S THAT
WHAT'S THIS IS GREATER THAN WHAT'S THAT

A$?WHAT'S THIS
B$?WHAT'S WHAT
WHAT'S THIS IS LESS THAN WHAT'S WHAT

A$?WHAT'S WHAT
B$?WHAT'S WHAT
WHAT'S WHAT IS EQUAL TO WHAT'S WHAT

A$?STOP
```

Figure 5-2. Execution of Program 5-4.

All of the comparison operators available for numeric comparison are available for string comparison.

We can manipulate strings in many ways. Consider the following statement:

```
200 C$ = A$ + B$
```

This does not perform numeric addition. Instead, it assigns a new string to the variable C$. The string variable assigned is the same string that would be displayed by the following PRINT statement:

```
200  PRINT A$;B$
```

We can enter a space in C$ in the following way:

```
200 C$ = A$ + " " + B$
```

This device might be used in a situation where A$ contains a person's first name and B$ contains the last name. To assign the name last name first we might use a statement such as

```
200 C$ = B$ + ", " + A$
```

This is called *concatenation* of strings. It is a very simple concept with a fancy name. When using concatenation there must not be more than 255 characters in the final string to be formed, or we will get a message:

```
?STRING TOO LONG ERROR IN 200
```

and our program will stop dead in its tracks. We get up to 255 characters without any special provision, and there is no way to get more in a single string variable. We can handle more characters by breaking the problem into segments each of which requires 255 or fewer characters.

.... SUMMARY

Applesoft provides string variables for storing character strings in a program. Strings may be assigned with INPUT, READ, and DATA, or the assignment statement. The maximum number of characters in a string is 255. Strings may be compared for order in an IF . . . THEN statement. Strings may be concatenated using a plus ("+") sign.

Problems for Section 5-1.1 .....

While many of these programs can be done in Integer BASIC, it is expected that you will do them in Applesoft.

1. Write a program that requests the user's name and responds with, "HELLO THERE 'YOUR NAME'", using the entered name where 'YOUR NAME' appears here.
2. Enter several words in DATA statements. Write a program that will display the data item that comes earliest in the alphabet. Be sure to use dummy data.
3. Enter several words in DATA statements. Write a program that will display only the word that is alphabetically last in the list.
4. Often in programs we want to ask the user questions for which only "YES" and "NO" are acceptable answers. Since we might want to do this at many points in the same program, it is useful to write one subroutine that sets a numeric variable to "1" for "YES" and "Ø" for "NO". Write such a subroutine.

5-1.2...Integer BASIC Strings

A string variable is distinguished from a numeric one by using a dollar sign ($) as the last character in the variable name.

We may work with string variables in many of the ways that we work with numeric variables. For instance, any of the following statements may appear in a program:

```
100  LET A$ = "FIRST"
100  INPUT A$
100  PRINT A$
```

A string may be empty. That is, it may contain no characters. The maximum number of characters is 25Ø. It turns out that the maximum number of characters really depends on the number of characters in our variable name. For a 1-character variable name we get 25Ø. We get 1 less character in the variable for each additional character in the variable

name. By some quirk, we can DIMension string variables up to 255, but we can never use them all. Strange things happen if we go past 250. It is a good idea to account for this problem right in your DIMension statements. If a string variable is to be used for more than 1 character it must be named in a DIMension statement. For example:

```
100 DIM A$(36),B$(12),C$(250),T1$(249)
```

will provide for 4 strings. A$ may contain up to 36 characters, B$ up to 12, C$ up to 250, and T1$ up to 249. Trying to DIMension a string for more than 255 characters will bring forth the

```
*** >255 ERR
```

error message. We may want to do something as simple as asking the player of a game to enter a name. Or we might use strings to allow the user to enter "YES" or "NO" instead of entering "1" for "YES" and "0" for "NO".

.... Double Subscript

Apple Integer BASIC strings allow us to access groups of characters and individual characters using subscripts. For example, if

```
A$="SUNMONTUEWEDTHUFRISAT"
```

then we can display "SUN" with the statement

```
200 PRINT A$(1,3)
```

Or we could PRINT A$(19,21) to display "SAT". A$(X,Y) defines all the characters from the Xth position to the Yth position in the string. Program 5-5 displays the names of the days of the week.

```
 80 REM    * DISPLAY THE DAYS OF THE WEEK
100 DIM A$(21)
120 A$="SUNMONTUEWEDTHUFRISAT"
140 FOR K=1 TO 7
150 J9=3*K-2
160 PRINT K,A$(J9,J9+2)
190 NEXT K
200 END
```

Program 5-5. Display the days of the week.

Look carefully at line 150. The idea here is to go 1, 4, 7, . . . 16, 19. Thus, on day 7 we get 3*7−2, which is 19, and we display the characters in positions 19 through 21.

```
>RUN
1         SUN
2         MON
3         TUE
4         WED
5         THU
6         FRI
7         SAT
```

Figure 5-3. Execution of Program 5-5.

Clearly, if A$(4,6) calls for characters 4, 5, and 6, then A$(7,7) calls for the 7th character. And A$(J,J) calls for the Jth character. Now we can access the characters of Integer BASIC strings individually.

.... Single Subscript

If we leave out the second subscript, then something quite different happens. Examine Program 5-6, which displays A$(K) for various values of K in line 13Ø.

```
    8Ø REM  * DEMONSTRATE THE USE OF A
       SINGLE SUBSCRIPT IN A STRING
   1ØØ DIM A$(25)
*  11Ø INPUT "?",A$
   12Ø FOR K=1 TO 25
*  13Ø PRINT A$(K)
   14Ø NEXT K
   15Ø END
```

Program 5-6. Single string subscript.

Note line 11Ø. There we request a string INPUT. In Integer BASIC the INPUT request for a string never displays a question mark; therefore, we have used prompted INPUT to display our own question mark.

We can display from a particular character of the string to the end with a single subscript. A$(1) calls for the entire string. A$(4) calls for the substring beginning with the fourth character and extending to the end. Thus, as the value of K increases by one for each step of the FOR . . . NEXT loop we get one less character at the beginning of the string.

```
>RUN
?HERE WE GO

HERE WE GO
ERE WE GO
RE WE GO
E WE GO
 WE GO
WE GO
```

```
E GO
 GO
GO
O
*** STRING ERR
STOPPED AT 130
```

Figure 5-4. Execution of Program 5-6.

Displaying A$(K) works well until we specify a value of K "off the end of the string." In Program 5-6 above, calling for the display of A$(11) when A$ contained only 10 characters caused the

```
*** STRING ERR
```

message that we see at the end of the program RUN. We can easily avoid this by using the LEN() function to measure the number of characters in a string.

.... The LEN() Function

LEN(A$) measures the number of characters actually stored in the string variable A$. Even though we may have dimensioned A$ to 250, the LEN() function will count only the number of characters that are really stored there. So, in our little program above, instead of having the FOR . . . NEXT loop in line 120 go from 1 to 25, we should code the following line:

```
120 FOR K=1 TO LEN(A$)
```

It is never good to allow a program to terminate in an error condition. This can cause other serious problems. Knowing the number of characters stored in a string variable enables us to tell the program when to stop.

.... String Comparison

Integer BASIC allows us to compare strings for equality in IF . . . THEN statements. The equals sign (=) is used to test "equals." The sharp sign (#) is used to test "not equals." Each of the following statements is valid.

```
100 IF A$=B$ THEN 135
100 IF A$#B$ THEN 240
100 IF A$(2,4)=A$(7,9) THEN END
100 IF T$(1,3)=S$(K,K+2) THEN 200
```

Using what we know at this point, we can display the characters of a string in alphabetical order. Suppose we enter the alphabet in a string constant so that we may compare each letter of the alphabet with each of the letters of some string entered from the keyboard during a program RUN. First we will look for As, then for Bs, and so on until we have looked for Zs. Each time we find that the current letter in the entered

string does not match the current letter of the alphabet, we skip to the next letter in the entered string. Each time we find a match, we print the matched character.

```
   90 REM  * OUR 1ST PROGRAM TO ALPHABETIZE CHARACTERS OF
      A STRING
  100 DIM AL$(26),B$(40)
  110 AL$="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
  120 PRINT "ENTER YOUR STRING:"
  130 INPUT "?",B$
  190 REM  * WE ALPHABETIZE BY USING A SAMPLE ALPHABET
  200 FOR K=1 TO 25
  210 FOR J=1 TO LEN(B$)
*  220 IF AL$(K,K)#B$(J,J) THEN 240
*  230 PRINT B$(J,J);
  240 NEXT J
  250 NEXT K
  900 END
```

Program 5-7. Alphabetizing in Integer BASIC.

The decision to display the current character is made in line 220. The actual display occurs in line 230. We might code lines 220 and 230 in the following single line:

```
220 IF AL$(K,K)=B$(J,J) THEN PRINT B$(J,J);

>RUN
ENTER YOUR STRING:
?BIRTHDAY

ABDHIRTY
```

Figure 5-5. Execution of Program 5-7.

Our little program seems to have done its job. But what will happen if we enter characters that are not letters of the alphabet? Let's try it.

```
>RUN
ENTER YOUR STRING:
?WHAT IS GOING ON HERE?

AEEGGHHIINNOORSTW
```

Figure 5-6. Another execution of Program 5-7.

We can see that any characters not included in AL$ are simply ignored. In many situations that is exactly what we would want.

There will be times when we will want to construct one string from another string or other strings. For instance, we might have a situation where the last name is stored in L$ and the first name is stored in F$ and

we want a new string, N$, to contain the entire name, first name first. It is a simple matter to set N$ equal to F$ with

```
140 N$=F$
```

.... Concatenation

Now, how do we get a space between the names? We use a statement of the following form.

```
150 N$( LEN(N$)+1)=" "
```

This statement appends a space to whatever is already in N$. And in a similar fashion we append the last name with

```
160 N$( LEN(N$)+1)=L$
```

The one thing to look out for is that the string N$ must be dimensioned large enough to accommodate all of the characters in F$, L$, and the space added in line 150. If you should fail to provide for this, don't worry; Integer BASIC will tell you about it with the

```
*** STR OVFL ERR
```

message. Simply make sure that your strings are adequately dimensioned for the job you intend to do. The process of adding characters to a string is called concatenation.

Suppose we have a name in N$ as described above. That is, N$="JOHN JONES". What would it take to write a program to create a new string containing the name, last name first, followed by a comma, a space, and the first name? All we have to do is find the space with a loop. Once we have found the space, it is a simple matter to rearrange the parts of the string in the desired order. Consider Program 5-8.

```
    90 REM * REARRANGE NAME FROM FIRST NAME FIRST TO LAST
       NAME FIRST
*  100 DIM N$(30),X$(31)
   110 PRINT
   120 PRINT "NAME - FIRST NAME FIRST"
   130 INPUT "ENTER HERE? ",N$
   135 IF N$="STOP" THEN 900
*  200 FOR I=1 TO LEN(N$)
   210 IF N$(I,I)=" " THEN 300
   220 NEXT I
   230 PRINT "ENTER A SPACE BETWEEN NAMES"
   240 GOTO 110
   290 REM
*  300 X$=N$(I+1)
*  310 X$( LEN(X$)+1)=", "
*  320 X$( LEN(X$)+1)=N$(1,I-1)
```

```
400 PRINT X$
410 GOTO 110
900 END
```

Program 5-8. Rearranging names in Integer BASIC strings.

By dimensioning X$ to one character more than N$ we guarantee enough space. Line 210 tests for the first space in N$. Then lines 300 through 320 rearrange the name string. See Figure 5-7.

```
>RUN

NAME - FIRST NAME FIRST
ENTER HERE? JOHN JONES

JONES, JOHN

NAME - FIRST NAME FIRST
ENTER HERE? STOP
```

Figure 5-7. Execution of Program 5-8.

There are a couple of things that could be done to improve our program. Suppose there is more than one space in the entered name. The program should reject it. Suppose someone enters last name first with a comma. The program should reject that also. It is left as an exercise to make these improvements.

....ASC()

Every character is stored in computer memory as a number. The numbers used by Integer BASIC are derived from the ASCII (American Standard Code for Information Interchange) character set. The values 193 through 218 are used for the letters "A" through "Z." We may learn what internal value Integer BASIC is using for any string character from the ASC() function.

ASC(A$) is the value used by Integer BASIC for the first character in the string variable A$. We can learn the value for the letter "T" by typing

```
>PRINT ASC("T")
```

Apple Integer BASIC will reply with 212. The actual values won't be important to us for most of our programs. The important concept here is that there is an order and that it places letters alphabetically in the correct sequence.

....What You Can't Do Directly in Integer BASIC (And How to Do It)

You can't assign to a string segment. A statement such as

```
200 A$(3,7)="FGHIJ"
```

is illegal and will bring forth the

```
*** SYNTAX ERR
```

message. Here is the way to perform the assignment above:

```
190 B$=A$(8)
200 A$(3)="FGHIJ"
210 A$(8)=B$
```

This assumes that A$ has at least eight characters and that B$ is adequately dimensioned.

You can't directly compare strings for order. The statement

```
200 IF A$<B$ THEN 300
```

is illegal. However, the ASC() function may be used to determine string order. ASC(A$(K,K)) and ASC(B$(J,J)) are in the same order as their corresponding characters. And since ASC() really refers to the first character, we can just as well code these as ASC(A$(K)) and ASC(B$(J)). Using this concept we can scan each of two strings one character at a time until we find a point where the character in one string is greater than or less than the character in the same position of the other string. When this happens we know the correct order. This is left as an exercise.

.... SUMMARY

String variables are designated by a trailing dollar sign ($). The number of characters a string variable may hold is up to 251 minus the length of your variable name. Any string that is to be used for more than a single character must be dimensioned. A$(I,J) names those characters from the Ith to the Jth inclusive. Coding A$(I) refers to all of those characters in A$ from I to the end. We may only assign characters to A$ or A$(I), where I may be any valid expression. The LEN() function returns the number of characters actually stored in the expression in parentheses. We may compare strings using either "equals" (=) or "not equal to" (≠) in an IF . . . THEN statement. The function ASC() gives us the internal numeric code used by Apple Integer BASIC for the first character in the expression in parentheses.

Problems for Section 5-1.2 .....

1. In Program 5-8, which reverses first and last names, make the improvements suggested. That is, check for a comma and check for extra spaces. Also note that if the user enters only the carriage return, the program will fail. Fix this, too.

2. We may determine ordering for two strings using the ASC() function to compare corresponding characters. Write a program to request two strings from the keyboard and display a message reporting whether the first is greater than, equal to, or less than the second. It is suggested that you test for equals before entering the loop that compares corresponding characters. Don't forget to account for strings of different lengths.
3. Write a program that accepts names, in the form last name first with a comma and a space between names, and displays them first name first.
4. Sometimes it is interesting simply to rearrange the contents of a string for display purposes. Write a program that enters the days of the week in a single string and displays them in the following format:

```
S   M   T   W   T   F   S
U   O   U   E   H   R   A
N   N   E   D   U   I   T
D   D   S   N   R   D   U
A   A   D   E   S   A   R
Y   Y   A   S   D   Y   D
        Y   D   A       A
            A   Y       Y
                Y
```

5. Write a program that displays the days of the week in the following format:

```
S   M   T   W   T   F   S
 U   O   U   E   H   R   A
  N   N   E   D   U   I   T
   D   D   S   N   R   D   U
    A   A   D   E   S   A   R
     Y   Y   A   S   D   Y   D
             Y   D   A       A
                 A   Y       Y
                     Y
```

6. Write a program that simply requests a string and displays it in reverse order.
* 7. In Problem 2 above we may not think to consider what happens if the user enters "6" and "12". Since "6" is greater than "1", it requires extra programming to place numbers in order when stored as strings. Write a string comparison routine to handle numbers like these correctly.

5-2.1...String Functions in Applesoft

A variety of string functions is available to make using strings in Applesoft very convenient. First, we list them all for easy reference.

```
ASC
CHR$
LEFT$
RIGHT$
MID$
LEN
STR$
VAL
```

....ASC

ASC is referred to as the "ASCII" function. ASC() returns a number from Ø to 255 that is derived from the ASCII (American Standard Code for Information Interchange) character set.

....CHR$

CHR$(X) becomes the character whose ASCII code is X. CHR$(9Ø) is Z, while the character for 32 is a space. The next time you get to an Apple, run Program 5-9.

```
100  FOR I = Ø TO 95
110  PRINT  CHR$(I);
120  NEXT I
```

Program 5-9. Display characters Ø through 95.

```
]RUN

 !"#$%&'()*+,-./Ø123456789:;<=>?@ABCDEFG
HIJKLMNOPQRSTUVWXYZ[\]^_
```

Figure 5-8. Execution of Program 5-9.

We see only 64 characters displayed even though the FOR loop calls for 96 characters. The first 32 characters are invisible and are therefore referred to as nonprinting. Several of these are of some interest. When we actually run Program 5-9, we will hear the Apple emit its beep. This occurs when CHR$(7) is output. We can produce the same result with CTRL-G. CHR$(8) corresponds to CTRL-H and is the same as the left arrow key that we use for program editing. CHR$(21) corresponds to CTRL-U and is the same as the right arrow key that we use for program editing. A right square bracket (the Applesoft prompt) appears in the display of our little program. We can get this character at the keyboard by typing SHIFT-M. The left square bracket ([), the back slash (\), and the underline (_) characters cannot be obtained from the keyboard. These 3 characters can be displayed only by printing their corresponding ASCII codes. Their ASCII codes are 91, 92, and 95 respectively.

....LEFT$
The LEFT$ function enables us to access the leftmost characters in a string. For example, LEFT$(A$,5) is the first 5 characters in A$. If there happen to be fewer than 5 characters stored in the string, then this expression represents full string. LEFT$(A$,X) represents the left X characters of A$ as long as the value of X is greater than Ø.

....RIGHT$
The RIGHT$ function is exactly analogous to the LEFT$ function, but for the right end of the string.

....MID$
To define characters within a string, use MID$. MID$(A$,X,Y) gives us the characters beginning with position X and continuing for Y characters. One way to describe the characters from position X and continuing through to the end of the string is with an expression such as MID$(A$,X). Note that this is not the same as RIGHT$(A$,X). We will create an error condition if we allow X to equal zero.

....LEN
LEN(A$) counts the number of characters actually stored in the string variable A$. LEN(X$) may be used anywhere a numeric expression is legal. For instance, we might code a line

```
1ØØ FOR X = 1 TO LEN(Y$)
```

if we want to perform some task for each character contained in the string Y$.

....STR$
The STR$ function converts a numeric value to string format. STR$(N) converts the internal binary code used to represent the numeric value of N into the ASCII code used for each of the digits. Let's examine the effect of a statement such as

```
2ØØ T$ =  STR$ (N)
```

While N stores a numeric value that we may command the computer to use in arithmetic calculations, T$ stores the digits of the number N as string characters. Thus T$ permits us to manipulate the digits using string functions of Applesoft.

....VAL
VAL is the reverse of the STR$ function. VAL(A$) converts the character string of digits in A$ into the binary format used for storing numbers. If the first character could not be part of a number, a "Ø" is returned. If the

function is successful in converting the beginning of a string, it continues until it finds an impossible character. When this happens VAL simply stops processing and returns the value up to that point. For example:

```
VAL("12 DAYS OF VACATION")
```

will convert to

```
12
```

This function handles scientific notation just fine. The value will be converted into the standard form for Applesoft. Thus:

```
VAL("123E-1")
```

will convert to

```
12.3
```

There they are: ASC, CHR$, LEFT$, LEN, MID$, RIGHT$, STR$, and VAL. Now let's use some of them.

Suppose we are working on a program to prepare financial reports. This means that we will be printing numbers that represent money in dollars and cents (or yuan and fen or whatever). Applesoft doesn't care what the units of our numeric values might be. As far as Applesoft is concerned, 1 dollar and 20 cents is 1.2, but we would like to show that as 1.20. So, our first task is to write a routine that will convert numeric values like 1.2 to string values like 1.20. We must also deal with values that come out to fractional cents. We must come up with a routine that will handle 381.2961 properly. Fundamentally, we are faced with a formatting problem.

Let's write this as a subroutine that accepts a number in M1 and returns a string in D$. Then we can easily write a little control routine to test it.

One way to make sure that a number like 1.2 has a trailing 0 is to multiply it by 100. So, 1.2 becomes 120. Of course, we must later insert the decimal point in the proper position. Our new number represents money in cents. Multiplying 381.2961 by 100 produces 38129.61. We need to round this off to the nearest cent. That can be done by adding .5 and eliminating the fractional portion of the resulting number. We saw in the last chapter that INT is made for just such a purpose. So, we may calculate the money values in cents with a statement such as

```
M9=INT(M1*100+.5)
```

Notice that we have left the value of M1 unchanged. It is a good idea to write subroutines that leave the input values intact.

Next, we can convert the number of cents from a numeric value to a string with

```
X$=STR$(M9)
```

Now, this string has no decimal point. We know that the two right digits represent cents and must appear to the right of a decimal point. Further, we know that the remaining digits represent dollars and must appear to the left of the decimal point. We may create the D$ string from these three pieces: dollars, decimal point, and cents. A decimal point may be included in one of two ways: enclose a decimal point in quotes or use CHR$(46). We find the code for a decimal point by printing ASC(".") in Applesoft. The number of digits in the dollar portion may be found using the LEN function:

```
D9=LEN(X$) - 2

Summing up:

Dollars       = LEFT$(X$,D9)
Decimal point = CHR$(46)
Cents         = RIGHT$(X$,2)
```

and all that remains is to build the output string by concatenating these three portions. See Program 5-1Ø.

```
   990  REM  * FORMAT DOLLARS AND CENTS
   1000 M9 =  INT (M1 * 100 + .5)
   1010 X$ =  STR$ (M9)
   1020 D9 =  LEN (X$) - 2
*  1030 D$ =  LEFT$ (X$,D9) +  CHR$ (46) +  RIGHT$ (X$,2)
   1040  RETURN
```

Program 5-1Ø. Formatting subroutine.

As mentioned earlier, we could have used "." instead of CHR$(46) in line 1Ø3Ø.

Now we can write a small control program to test our subroutine. This will require an INPUT statement to enter test values with some dummy value to terminate and a PRINT statement to display results. See Program 5-11.

```
90  REM  * TEST FORMATTER
100  INPUT "TEST VALUE? ";M1
110  IF M1 = -9999 THEN  END
120  GOSUB 1000
130  PRINT M1;" = ";D$
140  PRINT
150  GOTO 100
```

Program 5-11. Control routine to test Program 5-1Ø.

It is a good idea to provide a special value of M1 that will allow us to exit

the program without having to enter CTRL-C or press the RESET key. The value −9999 serves that purpose in this program.

```
]RUN
TEST VALUE? 1.2
1.2 = 1.20

TEST VALUE? -381.2961
-381.2961 = -381.30

TEST VALUE? 19
19 = 19.00

TEST VALUE? 381.29499
381.29499 = 381.29

TEST VALUE? -9999
```

Figure 5-9. Execution of Program 5-11.

Our program works well for the sample input values. However, consider what happens if the value of M1 is less than one dollar. How could we add a dollar sign? How could we put commas in to mark off thousands? Accountants like to put negative numbers in brackets. How could we do this? Some of these things are left as problems.

.... SUMMARY

The string functions ASC, CHR$, LEFT$, RIGHT$, MID$, LEN, STR$, and VAL have been presented. ASC(A$) returns the numeric code for the first character in A$, and CHR$(A) returns the character whose code is A. LEFT$, RIGHT$, and MID$ provide access to portions of strings. LEN(A$) returns the number of characters in A$. STR$(A) converts the numeric value of A to the string characters required to display it, and VAL(A$) converts the displayed characters to numeric representation.

Problems for Section 5-2.1 .....

1. Modify Program 5-10 to handle amounts less than one dollar.
2. Modify Program 5-10 to place a $ to the left of the first digit in the formatted result.
3. Modify Program 5-10 to insert commas to mark off thousands.
4. Correct Program 5-10 to properly display 0.00 if the amount is 0.
5. Modify Program 5-10 to enclose negative values in angle brackets. That is, −1.43 should display as ⟨1.43⟩.

* 6. Write a program to perform the reverse conversion. The string ⟨$1,234.51⟩, should convert to the numeric value −1234.51.

Hint: You'll want to use a FOR . . . NEXT loop and the MID$ function to pick out all of the possible special characters.

7. Problems 1 to 5 could be worked cumulatively. The result could be a program that performs all of the tasks described in the 5 problems. Do this.
8. Our formatter is a special case. It works only with hundredths. Extend this program to allow the user to specify the number of decimal places desired.
9. Given the date in yy/mm/dd form, display the date as Month dd, 19yy. That is, 82/12/31 becomes December 31, 1982. You may want to test for bad dates like 82/Ø4/31.
10. Write a program to display messages on the screen so that they scroll horizontally across the screen. Use DATA to supply the messages.
11. Project: Write a program to justify text by inserting spaces between words to fill a specified line width.

PROGRAMMER'S CORNER 5

BASIC Character Sets .....

....Applesoft Character Set

Experiment a little with the character set used by Applesoft. Program 5-9 was written to display only the characters associated with codes Ø through 95. Change that program to display the characters from Ø to 255. You will note that some characters are repeated more often than others. If you have a printer that can print lowercase letters, run your program so that the output goes to the printer.

....Integer BASIC Character Set

There is no CHR$ function in Integer BASIC. Let's not be deterred by that omission. Apple uses memory-mapped video for its display. That means that part of the computer's memory is used for display purposes. We can easily demonstrate that. Memory from 1Ø24 to 2Ø47 with the exception of 64 special bytes is used for the normal text-display screen. This is the same memory that is used for Lo-Res graphics. If we POKE values in this range of memory, we should see something on the screen. See Appendix C for more about POKE. If we

```
POKE 1Ø24,193
```

the letter "A" will appear in the upper left-hand corner of the screen—but

not if we are on the last line of the screen at the time. What happens then is that the "A" appears and is scrolled off the screen so fast that we don't see it. So move the cursor up the screen a little with the ESC-D sequence and try again. Try POKEing zero. That is an inverse @ sign. Try the following program:

```
100 FOR I=0 TO 255
110 POKE 1024,I
120 NEXT I
130 END
```

The display goes too fast to follow. Let's add

```
>115 FOR J=1 TO 500 : NEXT J
```

Now the display goes slowly enough for us to watch it. It would make a lot of sense to display the whole character set at once. We could take out line 115 and replace line 110 with

```
110 POKE 1024+I,I
```

And running that gives us a clue that the text-display screen is not laid out in one contiguous sequence of bytes. The first 40 bytes are fine, the second 40 bytes appear about one-third of the way down the screen, and the third 40 bytes appear about two-thirds of the way down the screen. Then the fourth 40 bytes fall into place right below the first 40. Studying the screen a little further reveals that there are 8 characters missing between the flashing 7 on the 3rd line of the display and the @ sign on the 4th line displayed, which appears on the 2nd line of the screen. Further, the 8 characters following the final conventional 7 in the lower right of the screen don't appear either. This is because the screen is laid out as shown in Figure 5-10 on the next page. For most of our work with the Apple we will be perfectly content to let the Apple wade through figuring out where to display things on its screen. But right now we've got a tiger by the tail, so let's pursue the text screen a bit more.

From Figure 5-10 we can see that the text screen is laid out in 3 display segments. Each is 40 characters across and contains 8 lines. Counting from 0, position 40 is displayed as the 1st character in line number 8. Position 80 is displayed as the 1st character in line number 16. And position 128 is displayed as the 1st character in line number 1. We saw that POKEing the 8 characters from 120 to 127 produced no visible display. That is because the Apple does not use the last 8 bytes of each group of 128 for display at all. Those bytes are saved for machine-language programmers. There are 8 groups of 128 bytes in the range from 1024 to 2047, so there are 8 groups of 8 or 64 bytes not used for display.

Now we can figure out what memory position corresponds to the leftmost character in each line. Which group of 128 memory bytes we're

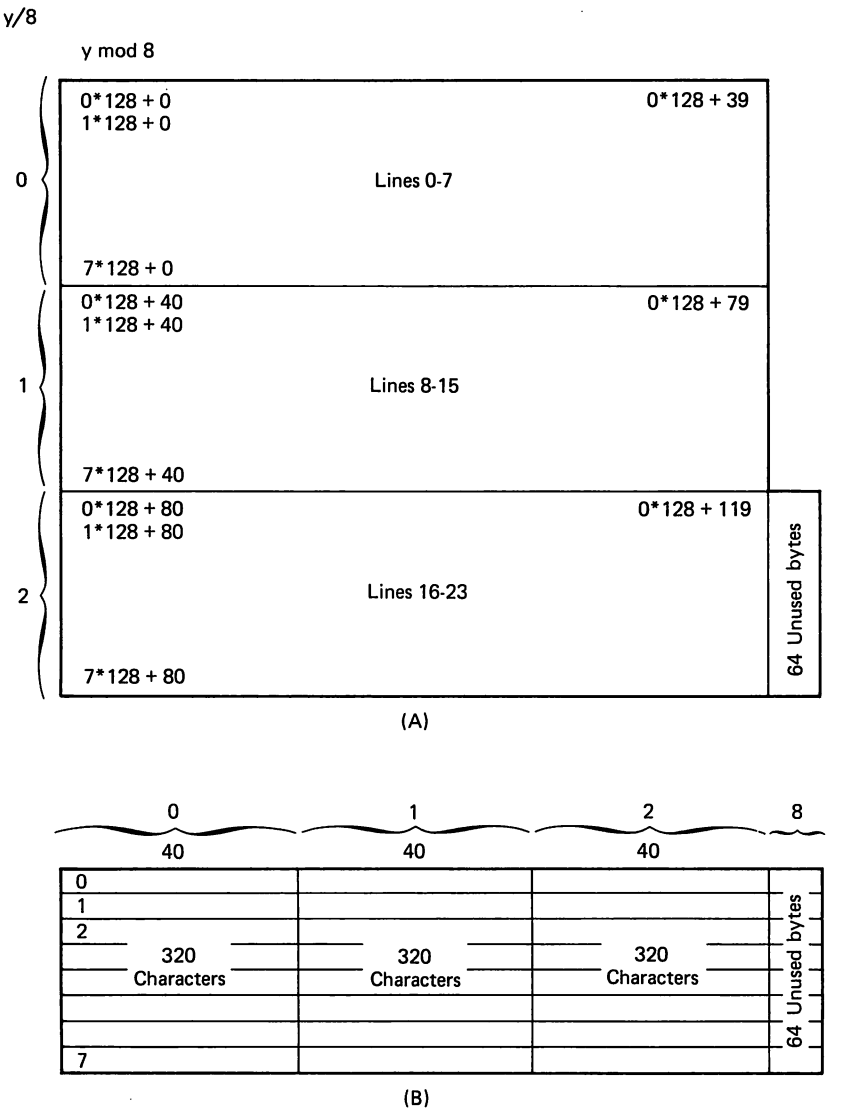


Figure 5-1Ø. The text screen on an Apple: (A) *The text screen.* (B) *Rearrange the text screen.*

on is the remainder after dividing by 8. That is Y MOD 8. Which group of 4Ø characters within the 128 is the integer part of Y divided by 8. In Integer BASIC that is Y/8. Remember that the 1st memory address of the display screen is 1Ø24. Thus the memory address of the line numbered Y may be found by the statement

```
2ØØ M=1Ø24 + (Y MOD 8)*128 + (Y/8)*4Ø
```

In fact, those parentheses are not required, but they make the statement a

little clearer for humans to read. We can easily write a little program to demonstrate this. See Program 5-12.

```
*  100 CALL -936
   102 FOR I=1 TO 10
   104 PRINT
   106 NEXT I
   150 FOR Y=0 TO 23
   200 M=1024+Y MOD 8*128+Y/8*40
   220 POKE M,Y
   250 NEXT Y
   300 END
```

Program 5-12. Demonstrate Integer BASIC display screen.

In line 100 we use a CALL as described in Appendix C to clear the screen and begin the display at the top. Then we have the program execute a few blank PRINT statements. This will move the cursor out of the way when the program terminates.

Now we can easily incorporate the necessary code to display the character set from 0 to 255. We simply start a counter at 0 and use an IF test to exit when we pass 255. See Program 5-13.

```
100 CALL -936
102 FOR I=1 TO 10
104 PRINT
106 NEXT I
110 C=0
150 FOR Y=0 TO 23
160 FOR X=0 TO 39
200 M=1024+Y MOD 8*128+Y/8*40+X
210 POKE M,C
220 C=C+1
230 IF C>255 THEN 300
240 NEXT X
250 NEXT Y
300 END
```

Program 5-13. Display Integer BASIC character set.

Here we have started with a position on the screen in terms of line number and position within line and calculated the corresponding location in memory for the Apple display screen. We have used a counter to terminate execution when we have POKEd 256 characters into the display screen.

Another approach would be to start with the position on the screen relative to the upper left-hand corner, calculate the line number and position within line, and then calculate the corresponding memory location.

Program 5-14 displays the character set for Integer BASIC using this last scheme.

```
100 CALL -936
102 FOR I=1 TO 10
104 PRINT
106 NEXT I
110 FOR C=0 TO 255
120 X=C MOD 40
130 Y=C/40
140 M=1024+Y MOD 8*128+Y/8*40+X
150 POKE M,C
160 NEXT C
300 END
```

Program 5-14. Display Ø to 255 with POKEs in Integer BASIC.

Chapter 6

Arrays

We have been using variables to store values one at a time. Such variables are referred to as "simple" variables. We have been able to perform marvelous feats on the computer with simple variables. We will accomplish even more with array variables. An array variable allows us to designate a collection of data values with a single variable name. Now, instead of designating the scores of the players in a five-player game with S1, S2, S3, S4, and S5, we may use an array variable. S(X) may be used to refer to the score of the Xth player. S(X) is read "S sub X". We may use the same variable name for an array as for a simple variable. You may want to avoid some confusion by not doing this, though. The value in parentheses is called a *subscript.* Each data value in the array is called an *element.* Using an array we could do the scoring for all five players with the same little segment of our BASIC program.

Arrays are used for storing information that naturally belongs together. Tax tables, pricing structures, inventory information, and life insurance premiums are all appropriate for using arrays. There are many times when an array is useful for storing information about the workings of the program itself. We may use arrays for storing test scores, temperatures, random numbers, and lists of all kinds. If we are working with Fibonacci numbers, it may be desirable to have them all in an array. (Remember them? They go: 1, 1, 2, 3, 5, 8, 13 . . .) Even though we might be able to re-create a particular sequence, it is convenient to have them all right there at the flick of a subscript.

6-1.1...Applesoft Numeric Arrays (One Dimension)

We may immediately benefit from the array concept by simply referring to array variables as needed. If we want the 6th element of T to be 5, we simply code a statement such as

```
200  T(5) = 5
```

We may readily use arrays in every way that we have been using simple variables. We may READ, PRINT, INPUT, and write IF . . . THEN tests using array variables. When an Applesoft program is executed, all elements of all arrays are set to zero.

In a given week we record the temperatures in Table 6-1.

Sunday	72
Monday	78
Tuesday	76
Wednesday	79
Thursday	85
Friday	85
Saturday	71

Table 6-1. Temperatures for a week.

There are any number of questions we might ask. We might want to know the average temperature or the highest and lowest temperatures. By using an array we can easily find the answers. Let's READ the data into elements one through seven of an array named W.

The average is easy. We just add up the seven temperatures and divide by seven. We may use T for the total. The first value of the total is the temperature for the first day.

We may find the highest and lowest temperatures by using two variables: H for high temp and L for low temp. Initially these may be set to the temperature of the first day, as it is at the same time the highest and lowest temperature.

The solutions for the three questions regarding temperatures each call for setting initial values and then performing some operation on each of the six days after the first—that is, Monday through Saturday. So our program will have a section to set up all of these initial values and a section with a loop that does some calculation for each of the three questions. See Program 6-1.

```
90  REM * ENTER THE TEMPERATURES IN ARRAY W
100  FOR J = 1 TO 7
110  READ W(J)
120  NEXT J
```

```
145  REM * SET UP INITIAL CONDITIONS
150 T = W(1)
160 H = W(1) : L = W(1)
190 :
200  FOR J = 2 TO 7
210 T = T + W(J)
230  IF W(J) > H THEN H = W(J)
240  IF W(J) < L THEN L = W(J)
250  NEXT J
290 :
300  PRINT "AVERAGE TEMP: ";T / 7
320  PRINT "HIGHEST TEMP: ";H
330  PRINT " LOWEST TEMP: ";L
890 :
900  DATA  72,78,76,79,85,85,71
990  END
```

Program 6-1. Find average, highest, and lowest temperatures.

```
]RUN
AVERAGE TEMP: 78
HIGHEST TEMP: 85
 LOWEST TEMP: 71
```

Figure 6-1. Execution of Program 6-1.

The next question that might be asked is, "How many times did the temperature increase, decrease, and remain unchanged?" We might now use the variables I, D, and U for this. We might want to know on what days the highest and lowest temperatures occurred. These are left as exercises.

Suppose we wish to simulate drawing numbers from a hat. We can easily do it with random numbers, provided that we may return each number to the hat before drawing the next one. If we must simulate drawing without replacement, then we must have a way of keeping track of what has been drawn. Here is an ideal application for an array. We simply set each element of an array equal to 1 and make the value Ø when that element has been selected. If the selected element is one then we know that it is available for use: use it and set it to Ø. If a selected element is Ø then we know that it is not available for use and we must select again. Let's look at such a program to draw 5 numbers at random from among 10. See Program 6-2.

```
90  REM * DRAWING FIVE NUMBERS AT RANDOM FROM AMONG TEN
95 :
100  FOR J = 1 TO 10
110 A(J) = 1
120  NEXT J
190 :
200  FOR J = 1 TO 5
210 R =  INT ( RND (1) * 10 + 1)
```

```
250  IF A(R) = Ø THEN 21Ø
260  PRINT "  ";R;
270 A(R) = Ø
280  NEXT J
290  PRINT
300  END
```

Program 6-2. Drawing five numbers at random from among ten.

```
]RUN
  2   6   1   3   8
```

Figure 6-2. Execution of Program 6-2.

From all appearances our program works just fine. It might be interesting to evaluate how well it does work. One measure of quality is the number of unusable random numbers generated. We can easily insert a counting variable to determine this. This is left as an exercise.

Considering the problem set before us, the trial-and-error method of the above program is not really a serious flaw in design. Drawing 5 numbers from among 10, or even drawing 10 from among 10, does not require major computer resources. However, what happens when we increase the numbers? Suppose we want to draw 1ØØ from among 1ØØ? It is worth investing some effort to eliminate the trial and error entirely.

Here is a plan that allows us to use every random number selected. First initialize the elements of the array as follows:

```
1ØØ  FOR J = 1 TO 1Ø
11Ø R(J) = J
12Ø  NEXT J
```

This means that each element stores one of the numbers in the range 1 to 1Ø. Next, select a random number in the range 1 to 1Ø and use that value as the subscript, say S. Now display R(S), replace R(S) with R(1Ø), and select a random number in the range 1 to 9. Since either we are on the 1st draw or we have replaced R(S), we will not need to determine if it has been used. (We know it has not been used.) Since we have moved R(1Ø) into a lower-numbered element, we may select from among fewer elements and still include all of the remaining numbers in the next random selection. The 2nd time through we move R(9) into the selected element. We simply repeat the select-display-replace sequence until the desired number of random draws have occurred.

We need to calculate the number of elements remaining. As the draw number (J) goes from 1 to 5, the number of elements remaining goes from 1Ø to 6. Thus, we can calculate the last element with

```
21Ø L = 1Ø - J + 1
```

See Program 6-3.

```
   90  REM * DRAWING RANDOM NUMBERS
        WITHOUT REPLACEMENT AND WITH
         NO TRIAL AND ERROR
  100   FOR J = 1 TO 10
  110 R(J) = J
  120   NEXT J
  190 :
  200   FOR J = 1 TO 5
*  210 L = 10 - J + 1
  230 S =  INT ( RND (1) * L + 1)
*  240   PRINT "  ";R(S);
*  250 R(S) = R(L)
  270   NEXT J
  290   PRINT
  300   END
```

Program 6-3. Drawing without replacement efficiently.

Notice that the element is printed in line 24Ø and then replaced in line 25Ø. L is always the number of active elements in the array. Even if we happen to select the Lth element, this method continues to function properly. The Lth element will be assigned to itself. No harm done.

```
]RUN
  6  2  4  8  1Ø
```

Figure 6-3. Execution of Program 6-3.

.... DIM

The highest subscript we have used is 1Ø. Whenever an array name is introduced, Applesoft automatically sets the highest subscript value to 1Ø. We may use the DIMension statement to set the highest subscript ourselves. We may want to do this to set either higher or lower limits.

```
100 DIM L(4),M(109),G3(1024)
```

This statement sets the highest subscript to 4 for array L, 1Ø9 for array M, and 1Ø24 for array G3.

Every Applesoft array we use allows the subscript to have a value of Ø. This is true whether or not the DIMension statement is used. Therefore, in the absence of a DIM we get 11 elements. In the sample statement above, L consists of 5 elements, M consists of 11Ø elements, and G3 provides for 1Ø25 numbers. Initially, when we have no particular need for the Ø element, we may simply ignore it.

.... SUMMARY

An array enables us to manage a number of variables using one variable name. DIM X(N) sets aside N+1 elements in an array named X. Array elements may be utilized in Applesoft program statements wherever a

simple numeric variable may be utilized (with the exception that array variables may not be used as the loop variable in FOR . . . NEXT). With arrays we will often find it convenient to use FOR . . . NEXT loops to process all elements or a block of elements.

Problems for Section 6-1.1 .....

1. Modify the daily-temperature program (Program 6-1) to tabulate the number of times the temperature increased, decreased, and remained unchanged.
2. Modify the daily-temperature program (Program 6-1) to determine on which days the highest and lowest temperatures occurred.
3. In the first program that draws numbers from a hat (Program 6-2) insert a variable to count the number of unusable numbers generated. Run the program several times to get a range of values.
4. Do Problem 3 drawing 1Ø from among 1Ø.
5. Modify Program 6-3 to select 1ØØ numbers from among 1ØØ.
6. Fill a 2Ø-element array with twice the value of the subscript. Display all of the elements in order and in reverse order.
7. Fill 1 array with the values 6, 3, and 9. Fill a 2nd array with the values 2, 8, 6, and 5. Display all possible pairs of 1 element from each array. There are 12 pairs.
8. Fill two arrays as in Problem 7. Fill a third array with all elements from these two arrays with no duplicates.
9. Fill a 1ØØ-element array with random numbers. Count the number of increases, decreases, and the number of no changes. Calculate the average.

6-1.2...Integer BASIC Arrays

The features of Integer BASIC arrays are similar in many ways to the features of Applesoft arrays as described in Section 6-1.1. So, you should read that section before reading this one.

In order to use an array variable in Integer BASIC we must code a DIMension statement before any reference to an element of the array. So:

```
1ØØ DIM A(5),B(17)
```

prepares for 5 elements in an array named A and 17 elements in a B array. Unlike Applesoft, Integer BASIC arrays do not permit Ø subscripts.

The computer array provides the ability to store many numbers so that we may process selected elements by knowing what subscript to use. Often all elements will be processed by using a FOR . . . NEXT loop.

.... Warning: Arrays Not Cleared in Integer BASIC

When a program is executed in Integer BASIC, the values in the array are not set to zero as in Applesoft. The values in the array will be whatever

happens to be in the memory cells used by the array at the time that the program is executed. Those values will be different from time to time depending on what the machine has been doing just prior to running this program. So, if we want an array to contain all zeros, then we must include a little routine that assigns zero to each element.

Integer BASIC does not provide for READ . . . DATA. We must enter data with either a series of assignment statements or INPUT.

Problems for Section 6-1.2

All data for these Integer BASIC programs may be entered from the keyboard.

1. Modify the daily-temperature program (Program 6-1) to tabulate the number of times the temperature increased, decreased, and remained unchanged.
2. Modify the daily-temperature program (Program 6-1) to determine on which days the highest and lowest temperatures occurred.
3. In the first program that draws numbers from a hat (Program 6-2) insert a variable to count the number of unusable numbers generated. Run the program several times to get a range of values.
4. Do Problem 3 drawing 10 from among 10.
5. Modify Program 6-3 to select 100 numbers from among 100.
6. Fill a 20-element array with twice the value of the subscript. Display all of the elements in order and in reverse order.
7. Fill 1 array with the values 6, 3, and 9. Fill a 2nd array with the values 2, 8, 6, and 5. Display all possible pairs of 1 element from each array. There are 12 pairs.
8. Fill two arrays as in Problem 7. Fill a third array with all elements from these two arrays with no duplicates.
9. Fill a 100-element array with random numbers. Count the number of increases, decreases, and the number of no changes. Calculate the average.

6-2...Applesoft Numeric Arrays (Multiple Dimension)

We have seen that single-dimension arrays may be used to organize data in a list. We may also use two or more subscripts to arrange data into tables of all kinds. We might be interested in the temperature at 6:00 A.M., 12:00 noon, and 6:00 P.M. for a week. For this we need an array with two subscripts. Such an array is referred to as two-dimensional. We will use one dimension to represent the days of the week and the other to represent the three different times of day. To do several weeks we might use a third

dimension. Let's look at a little program to find the average daily temperature using three readings a day. See Program 6-4.

```
   90  REM  * FIND AVERAGE TEMP
   100  FOR DA = 1 TO 7
   110  FOR RE = 1 TO 3
*  120  READ TE(DA,RE)
   130  NEXT RE
   140  NEXT DA
   175 :
   180  PRINT "       TEMPERATURE"
   190  PRINT "DAY 6AM 12N 6PM AVG"
   200  FOR DA = 1 TO 7
   202  PRINT DA;"    ";
   205 T = 0
   210  FOR RE = 1 TO 3
   220 T = T + TE(DA,RE)
   230  PRINT TE(DA,RE);"   ";
   240  NEXT RE
   250  PRINT T / 3
   260  NEXT DA
   980 :
  1000  DATA  76,79,75, 72,77,76
  1020  DATA  74,79,81, 75,80,83
  1040  DATA  80,77,70, 68,65,65
  1060  DATA  65,67,76
```

Program 6-4. Find daily average temperature.

By naming 2 subscripts in line 120 we caused Applesoft to allow automatically for 11 elements in each dimension. Since we only require values of up to 7 in one dimension and 3 in the other, we would use the statement

```
95  DIM TE(7,3)
```

It is good practice to include the DIMension statement at the beginning of every program even if it is not required for our application. The DIMension statement reveals something about our program to the reader. Even if we want an array DIMensioned to (10, 10), we should do so with a DIMension statement. In the absence of the DIMension statement, the reader doesn't know that we are using an array until it appears in a statement of the program. Even then the reader has no idea how much of the array we are using.

```
]RUN
       TEMPERATURE
DAY 6AM 12N 6PM AVG
1   76  79  75  76.6666667
2   72  77  76  75
3   74  79  81  78
```

```
4    75   80   83   79.3333334
5    80   77   70   75.6666667
6    68   65   65   66
7    65   67   76   69.3333334
```

Figure 6-4. Execution of Program 6-4.

.... Zero Subscripts

The zero subscript is always available. In many programming situations the zero subscript is a great convenience. The zero term of a polynomial is easily represented in this way. The positions reserved for the zero subscripts are there whether we use them or not. For most programs the impact of zero subscripts is minor. However, when writing large programs on a machine with little memory, it may become necessary to use them just to get the program to fit.

.... More Than Two Subscripts

Applesoft will accommodate up to 88 subscripts. Yes, as long as most of the dimensions are Ø, we can do this. In practice we are unlikely to require so many. If we think we need 88, we probably are taking an inefficient approach to solving our problem. Three dimensions are often very convenient. We should always include the DIMension statement at the beginning of the program. For more than 3 dimensions we *must* include it, since a real array 11 by 11 by 11 by 11 won't even fit in a 48K machine. Would you believe a 71.5K machine? Not only must we provide a DIMension, but it must call for a smaller array than that.

.... SUMMARY

We have multidimensional arrays in Applesoft. D(3,4) refers to the value in column 4 of row 3. Since Ø subscripts are included, column 4 is actually the 5th column and row 3 is actually the 4th row. We are not required to use Ø subscripts, but using them will conserve memory.

As with single-dimension arrays, the DIMension statement specifies the maximum subscript in each dimension.

```
100  DIM X(5,3,8)
```

prepares for an array of 3 dimensions 7 by 4 by 9. Often we process data in arrays with loops and nested loops. Even though Applesoft automatically provides 11 elements in each dimension, we should always include the DIMension statement to help document our program.

Problems for Section 6-2 .....

1. In Program 6-4, find the maximum temperature for each of the three reading times (6:00 A.M., 12:00 noon, and 6:00 P.M.).

2. In Program 6-4, find the maximum temperature for each day.
3. In Program 6-4, find the average temperature for each of the three reading times (6:00 A.M., 12:00 noon, and 6:00 P.M.).
4. Fill two 4-by-5 arrays with random numbers and display them. Then fill a 3rd array with the sums of the corresponding entries from the first 2 arrays and display the result.
* 5. In a 10-by-10 array enter all 1s in the upper left to lower right diagonal and the leftmost column, and all zeros elsewhere. Then beginning in the 3rd row, 2nd column, enter the sum of the entry in the same column of the row immediately above and in the column 1 to the left and the row immediately above, through the 10th row, 9th column. That is:

```
230 P(R,C) = P(R-1,C) + P(R-1,C-1)
```

for the described range. Display the resulting array.

6-3...Applesoft String Arrays

The ability to use arrays to store alphabetic data is very nice. The relationship between string simple variables and string arrays is exactly analogous to the relationship between numeric simple variables and numeric arrays. Each string array consists of a collection of string elements all referred to by the same array variable name with a subscript.

Each element of the string array has the same properties as a string simple variable. Each element may store up to 255 characters. We may READ, INPUT, assign, and PRINT elements of string arrays. And we may apply all of the string functions discussed in Chapter 5: ASC, CHR$, LEFT$, RIGHT$, MID$, LEN, STR$, and VAL. Let's see the convenience of using string arrays for labeling. Program 6-5 READs the names of the days of the week into an array and then displays them.

```
90  REM  * READ AND DISPLAY DAYS OF THE WEEK
95  DIM W$(7)
100  FOR DA = 1 TO 7
110  READ W$(DA)
120  NEXT DA
190 :
200  FOR DA = 1 TO 7
210  PRINT W$(DA)
220  NEXT DA
990 :
1000  DATA  SUNDAY
1010  DATA  MONDAY
1020  DATA  TUESDAY
1030  DATA  WEDNESDAY
1040  DATA  THURSDAY
1050  DATA  FRIDAY
1060  DATA  SATURDAY
```

Program 6-5. Display the days of the week.

```
]RUN
SUNDAY
MONDAY
TUESDAY
WEDNESDAY
THURSDAY
FRIDAY
SATURDAY
```

Figure 6-5. Execution of Program 6-5.

Once the string data is stored in the elements of the string array, we may manipulate it in many ways. It may be that on a report we want the days of the week spelled out in one place and abbreviated in another. We can easily do this with the LEFT$ function. We can demonstrate this with a simple change in line 21Ø.

```
21Ø PRINT LEFT$(W$(DA),3);"  ";W$(DA)

]RUN
SUN  SUNDAY
MON  MONDAY
TUE  TUESDAY
WED  WEDNESDAY
THU  THURSDAY
FRI  FRIDAY
SAT  SATURDAY
```

Figure 6-6. Execution of modified Program 6-5.

Recall that in Program 6-4 to average the 3 temperatures taken each day for a week we labeled the days of the week from 1 to 7. We now have the ability to produce a more readable report. We may modify that program to label each line with the weekday name. If we use the full day names, then we have to deal with the fact that not all names have the same number of letters. We can handle this by using comma spacing, but then we are forced to place the names in a display field of 16 characters. That seems like too much space. The longest name contains 9 letters. The easy way out for now is to abbreviate. Let's do it this way. See Program 6-6.

```
  9Ø  REM  * FIND AVERAGE TEMP
  95  DIM W$(7),TE(7,3)
  1ØØ  FOR DA = 1 TO 7
* 1Ø5  READ W$(DA)
  11Ø  FOR RE = 1 TO 3
  12Ø  READ TE(DA,RE)
  13Ø  NEXT RE
  14Ø  NEXT DA
  175 :
  18Ø  PRINT "       TEMPERATURE"
  19Ø  PRINT "DAY  6AM 12N 6PM AVERAGE"
```

```
   200  FOR DA = 1 TO 7
*  202  PRINT  LEFT$ (W$(DA),3);"   ";
   205 T = 0
   210  FOR RE = 1 TO 3
   220 T = T + TE(DA,RE)
   230  PRINT TE(DA,RE);"   ";
   240  NEXT RE
   250  PRINT T / 3
   270  NEXT DA
   990 :
   1000  DATA  SUNDAY,76,79,75
   1010  DATA  MONDAY,72,77,76
   1020  DATA  TUESDAY,74,79,81
   1030  DATA  WEDNESDAY,75,80,83
   1040  DATA  THURSDAY,80,77,70
   1050  DATA  FRIDAY,68,65,65
   1060  DATA  SATURDAY,65,67,76
```

Program 6-6. Display average daily temperature with day names.

Look at the DATA section. We have included the days of the week right in with the temperature data. Doing it this way helps to clearly document which temperatures go with which day.

```
]RUN
      TEMPERATURE
DAY   6AM  12N  6PM  AVERAGE
SUN   76   79   75   76.6666667
MON   72   77   76   75
TUE   74   79   81   78
WED   75   80   83   79.3333334
THU   80   77   70   75.6666667
FRI   68   65   65   66
SAT   65   67   76   69.3333334
```

Figure 6-7. Execution of Program 6-6.

This report is easy to read. We do not wonder whether day 1 is Sunday or Monday. Four of the averages are displayed with 9 digits. We might want to round those values off to the nearest 10th. If it is important to have the day names spelled out, then we could easily change the appearance of Figure 6-7 using TAB or HTAB in Program 6-6.

Suppose we have a record store and are using a computer to help calculate sales slips for us. Each record is marked with a letter H through P. This letter is assigned according to the price of the record. Thus, H is the label on every $2.99 record, and I is the label on every $3.45 record. We can easily write a program using arrays to calculate a total sale for us.

We can enter the correspondence between letters and prices into the program by READing DATA. Two arrays will be required—one string array for the letter codes, and one numeric array for the prices. It is a

simple matter to arrange the data so that the letter codes and the prices are properly coordinated. Placing the data in DATA statements makes it a simple matter to add new codes or change prices. We will use "STOP" as the signal to stop reading data. It is always a good idea to leave a gap in line numbers between the real data and the termination signal. See Program 6-7.

```
90  REM  * CALCULATE SALES SLIPS
100  DIM N$(26),P(26)
200  FOR I = 1 TO 26
210  READ N$(I),P(I)
220  IF N$(I) = "STOP" THEN 250
230  NEXT I
250 N1 = I - 1
285 :
290  REM  * REQUEST INPUT AND CALCULATE HERE
300  PRINT "('END' TO STOP)"
310 T = 0:N = 0
320  PRINT "RECORD: ";
330  INPUT R$
335  IF R$ = "END" THEN 500
340  FOR J = 1 TO N1
350  IF R$ = N$(J) THEN 400
360  NEXT J
370  PRINT "NOT FOUND - REENTER"
380  GOTO 320
400 T = T + P(J)
410 N = N + 1
420  GOTO 320
490 :
500  PRINT
510  PRINT "RECORDS: ";N
520  PRINT "TOTAL: $";T
900  END
990 :
1000  DATA  H,2.99,  I,3.45
1010  DATA  J,3.69,  K,3.99
1020  DATA  L,4.49,  M,4.99
1030  DATA  N,5.99,  O,6.99
1040  DATA  P,7.99
1190  DATA  STOP,0
```

Program 6-7. Total price in record store.

Program 6-7 is set up in 4 segments. The 1st segment from 100 to 250 reads in the price data. The 2nd segment from 300 to 420 handles the entry of figures for each sale. Lines 500 to 520 display the final results. And the 4th segment is the DATA in lines 1000 to 1190.

```
]RUN
('END' TO STOP)
RECORD: ?H
RECORD: ?P
RECORD: ?P
RECORD: ?O
RECORD: ?L
RECORD: ?A
NOT FOUND - REENTER
RECORD: ?END

RECORDS: 5
TOTAL: $3Ø.45
```

Figure 6-8. Execution of Program 6-7.

.... Geography

Let's write a program to play Geography—a simple game for two or more players. We will write a program for a person to compete with the computer. Each player says the name of a place such that the first letter is the same as the last letter of the name chosen by the previous player. Of course, the first name can be any place at all. If I say BOSTON, then you might say NEW YORK. That fits the rule, because BOSTON ends with an "N" and NEW YORK begins with an "N". The next player might think of KANSAS. No name may be used a second time. The first person unable to think of an appropriate name drops out.

We can easily program the computer so that it "remembers" all of the names used. The more games the computer plays, the tougher it will be to beat.

We need a string array to hold all of the names. We can use a numeric array to tell us if a specific name has been used. Let's set up a numeric array AV() so that a one (1) indicates that the name in the corresponding position of the NA$() names array is available for use and a zero (Ø) means that the name has been used in this game. If AV(5) = 1, then NA$(5) may be used. We can enter a few names into the NA$() array using DATA statements. This way the computer has some names to start with. Let's allow the computer to produce the first name.

It may sound like a big job to produce a program that performs as described. We can easily trim the job down to size by spending a little extra time organizing before we generate any BASIC program statements. Think about the steps in the game. There are six easily defined segments in our program.

1. Read the names into the NA$ array.
2. Display the instructions.
3. Initialize the AV array to all ones.
4. Have the computer begin the game.

5. Process the person response.
6. Prepare the computer response.

Each of these six jobs may be programmed as a subroutine. The advantages of doing it this way are tremendous. When we first test our completed program it will be easy to spot which subroutine is not performing properly. Once we are satisfied that our program is working well, it will be a simple matter to determine which subroutines we need to modify or replace to change the program so that the names are stored in a file on disk.

Let's begin by writing the control routine that will manage the 6 subroutines listed above. In thinking about this routine we need to handle the situation when the computer runs out of names in number 6. We can save the computer response in a string variable and save "QUIT" when the computer quits. This thought leads us to think about letting the person quit at any time. Thus we select CP$ for the computer response and PE$ for the person response. Further, we may give the player the option to play another game. We arbitrarily decide to provide for 300 names. See Program 6-8a.

```
20  DIM NA$(300),AV(300)
30  GOSUB 8000 : REM  * READ NAMES ARRAY
35  GOSUB 9000 : REM  * INSTRUCTIONS
37  GOSUB 4000 : REM  * INITIALIZE AVAILABLE NAMES ARRAY
40  GOSUB 7000 : REM  * COMPUTER STARTS
50  GOSUB 6000 : REM  * PERSON RESPONDS
58  IF PE$ = "QUIT" THEN 75
60  GOSUB 5000 : REM  * RESPONSE OF COMPUTER
65  IF CP$ <  > "QUIT" THEN 50
75  PRINT "DO YOU WANT ANOTHER GAME";
80  INPUT A$
85  TEXT
90  IF LEFT$(A$,1) = "N" THEN  END
100  FOR I9 = 1 TO 1000 : NEXT I9
120  GOTO 35
```

Program 6-8a. Control routine to play Geography.

The six steps have become six subroutines at lines 8000, 9000, 4000, 7000, 6000, and 5000. The choice of line numbers is arbitrary. Now we are well prepared to write each individual subroutine.

We read the names at 8000. The place names are entered in DATA statements. We choose to provide the signal data "DONE". See Program 6-8b.

```
7996 :
7998  REM * READ NAMES
8000 I9 = 1
8010  READ NA$(I9)
```

```
   8020  IF NA$(I9) = "DONE" THEN 8080
   8030 I9 = I9 + 1 : GOTO 8010
*  8080 N0 = I9 - 1
   8090  RETURN
   8096 :
   8100  DATA  NEW YORK, CHICAGO, PHILADELPHIA, BOSTON
   8590  DATA  "DONE"
```

Program 6-8b. Read names into an array for Geography game.

Notice that line 8080 saves the number of names in the array in numeric variable N0.

Instructions are simple enough. We can just display a little description on the screen. Think about that. How fast do people read? We must provide a way for the fast reader to move on and allow the slow reader a chance to finish. We can do this by asking the person to tell the program when he or she is ready. See Program 6-8c.

```
   8996 :
   8998  REM  * INSTRUCTIONS
   9000  TEXT : HOME
   9005  PRINT "THIS PROGRAM WILL PLAY A GEOGRAPHY GAME" :
         PRINT
   9010  PRINT "WITH YOU.  YOU WILL TAKE TURNS WITH THE" :
         PRINT
   9015  PRINT "COMPUTER.  EACH OF YOU WILL BE TRYING TO"; :
         PRINT
   9020  PRINT "THINK OF NAMES OF PLACES SUCH THAT THE" :
         PRINT
   9025  PRINT "FIRST LETTER OF YOUR NAME IS THE SAME AS"; :
         PRINT
   9030  PRINT "THE LAST LETTER OF THE PREVIOUSLY USED" :
         PRINT
   9035  PRINT "PLACE NAME." : PRINT
*  9040  POKE 34,15
   9045  HOME : INPUT "ARE YOU READY? ";A$
   9065  IF LEFT$(A$,1) <  > "Y" THEN 9045
   9070  FOR I9 = 1 TO 1000 : NEXT I9
*  9080  TEXT : HOME
   9090  RETURN
```

Program 6-8c. Geography-game instructions.

The wording of instructions is somewhat subjective. Instructions should tell the user what to expect. That POKE 34,15 at line 9040 sets the top of the text window at line 15 so that we can freeze the instructions on the screen. TEXT at line 9080 undoes the POKE at line 9040.

The initialization of the AV array beginning at line 4000 is very straightforward. See Program 6-8d.

```
3996 :
3998  REM  * INITIALIZE AVAILABLE NAMES ARRAY
4000  FOR J9 = 1 TO N0
4010 AV(J9) = 1
4020  NEXT J9
4090  RETURN
```

Program 6-8d. Initialize available-names array.

To start the game at line 7000 we have the computer select at random a name from the names array. The place must be recorded as used and the CP$ string variable is loaded with the name selected. See Program 6-8e.

```
6996 :
6998  REM  * COMPUTER BEGIN THE GAME
7000 X9 =  INT ( RND (1) * N0 + 1)
7020 CP$ = NA$(X9) : AV(X9) = 0
7030  PRINT "FIRST PLACE : ";CP$
7090  RETURN
```

Program 6-8e. Begin Geography game.

Once the computer has produced a place name, the program proceeds to the person-response subroutine.

We agreed to have the person response stored in PE$. The person response must pass a number of tests. It ought to have at least two characters. That is handled with the LEN() function. The first letter of the person response must match the last letter of the computer place name. We do that with the RIGHT$() and LEFT$() string functions. If PE$ passes these two tests then we must see if it is in the list of names stored in the NA$() array. If PE$ is in the list, has it been used during this latest game? If it is not in the list, then we put it in the list. See Program 6-8f.

```
5996 :
5998  REM  * PERSON GO
6000  PRINT
6010  INPUT "   YOUR TURN: ";PE$
6012  IF PE$ = "QUIT" THEN 6190
6015  IF LEN (PE$) > 1 THEN 6030
6020  PRINT "NAME TOO SHORT" : GOTO 6010
6030  IF LEFT$(PE$,1) = RIGHT$(CP$,1) THEN 6040
6035  PRINT "NO MATCH" : GOTO 6010
6040  FOR I9 = 1 TO N0
6045  IF PE$ = NA$(I9) THEN 6100
6050  NEXT I9
6055  IF N0 < 300 THEN  6065
*  6060  PRINT "NO ROOM FOR MORE NAMES" : GOTO 6010
6065 N0 = N0 + 1
6070 NA$(N0) = PE$ : AV(N0) = 0
```

```
6080  GOTO 6190
6096 :
6098  REM * "FOUND NAME"
6100  IF AV(I9) = 1 THEN 6150
6110  PRINT "USED ALREADY" : GOTO 6010
6150 AV(I9) = 0
6190  RETURN
```

Program 6-8f. Person-response subroutine in Geography.

In the unlikely event that someone runs enough games to build the names array up to 300 names, line 6060 of this subroutine will display a message and request another name.

Finally, the computer-response subroutine at line 5000 completes the program. We simply search the NA$() array for a place name that has the proper first letter and that has not been used in this latest game. If no such name is found, save the word "QUIT" in CP$. See Program 6-8g.

```
4996 :
4998  REM  * COMPUTER RESPOND
5000  FOR I9 = 1 TO N0
5010  IF LEFT$(NA$(I9),1) = RIGHT$(PE$,1) AND AV(I9)
      = 1 THEN 5050
5015  NEXT I9
5020  PRINT : PRINT " I HAVE RUN OUT OF NAMES"
5025 CP$ = "QUIT"
5030  GOTO 5090
5050 CP$ = NA$(I9) : AV(I9) = 0
5060  PRINT "     I CHOOSE: ";CP$
5090  RETURN
```

Program 6-8g. Computer-response subroutine for Geography.

The program does not verify that the names are actually legitimate place names. That is left to the honor of the player. This same program allows the player to change the rules of the game. We could just as well do people's names or a computer glossary. In that case, we would want to change the instructions and the DATA. Notice that in the computer-response subroutine at line 5000 the entire list is scanned for names. Since every name that is added to the list during the game is by definition not available for the remainder of this game, the program need not do this. We could establish another variable to hold the number of names at the beginning of the current game. We could also have the computer begin at a random place in the NA$() array instead of beginning with the first name every time. This change would add variety to the game.

We list the complete program here for your convenience.

```
   20  DIM NA$(300),AV(300)
   30  GOSUB 8000 : REM  * READ NAMES ARRAY
   35  GOSUB 9000 : REM  * INSTRUCTIONS
   37  GOSUB 4000 : REM  * INITIALIZE AVAILABLE NAMES ARRAY
   40  GOSUB 7000 : REM  * COMPUTER STARTS
   50  GOSUB 6000 : REM  * PERSON RESPONDS
   58  IF PE$ = "QUIT" THEN 75
   60  GOSUB 5000 : REM  * RESPONSE OF COMPUTER
   65  IF CP$ <  > "QUIT" THEN 50
   75  PRINT "DO YOU WANT ANOTHER GAME";
   80  INPUT A$
   85  TEXT
   90  IF LEFT$(A$,1) = "N" THEN  END
  100   FOR I9 = 1 TO 1000 : NEXT I9
  120   GOTO 35
 3996 :
 3998   REM  * INITIALIZE AVAILABLE NAMES ARRAY
 4000   FOR J9 = 1 TO N0
 4010 AV(J9) = 1
 4020   NEXT J9
 4090   RETURN
 4996 :
 4998   REM  * COMPUTER RESPOND
 5000   FOR I9 = 1 TO N0
 5010   IF LEFT$(NA$(I9),1) = RIGHT$(PE$,1) AND AV(I9) = 1
        THEN 5050
 5015   NEXT I9
 5020   PRINT : PRINT " I HAVE RUN OUT OF NAMES"
 5025 CP$ = "QUIT"
 5030   GOTO 5090
 5050 CP$ = NA$(I9) : AV(I9) = 0
 5060   PRINT "     I CHOOSE: ";CP$
 5090   RETURN
 5996 :
 5998   REM  * PERSON GO
 6000   PRINT
 6010   INPUT "   YOUR TURN: ";PE$
 6012   IF PE$ = "QUIT" THEN 6190
 6015   IF LEN (PE$) > 1 THEN 6030
 6020   PRINT "NAME TOO SHORT" : GOTO 6010
 6030   IF LEFT$(PE$,1) = RIGHT$(CP$,1) THEN 6040
 6035   PRINT "NO MATCH" : GOTO 6010
 6040   FOR I9 = 1 TO N0
 6045   IF PE$ = NA$(I9) THEN 6100
 6050   NEXT I9
 6055   IF N0 < 300 THEN  6065
*6060   PRINT "NO ROOM FOR MORE NAMES" : GOTO 6010
 6065 N0 = N0 + 1
 6070 NA$(N0) = PE$ : AV(N0) = 0
 6080   GOTO 6190
 6096 :
 6098   REM * "FOUND NAME"
```

```
   6100  IF AV(I9) = 1 THEN 6150
   6110  PRINT "USED ALREADY" : GOTO 6010
   6150 AV(I9) = 0
   6190  RETURN
   6996 :
   6998  REM  * COMPUTER BEGIN THE GAME
   7000 X9 =  INT ( RND (1) * N0 + 1)
   7020 CP$ = NA$(X9) : AV(X9) = 0
   7030  PRINT "FIRST PLACE : ";CP$
   7090  RETURN
   7996 :
   7998  REM * READ NAMES
   8000 I9 = 1
   8010  READ NA$(I9)
   8020  IF NA$(I9) = "DONE" THEN 8080
   8030 I9 = I9 + 1 : GOTO 8010
*  8080 N0 = I9 - 1
   8090  RETURN
   8096 :
   8100  DATA  NEW YORK, CHICAGO, PHILADELPHIA, BOSTON
   8590  DATA  "DONE"
   8996 :
   8998  REM  * INSTRUCTIONS
   9000  TEXT : HOME
   9005  PRINT "THIS PROGRAM WILL PLAY A GEOGRAPHY GAME" :
         PRINT
   9010  PRINT "WITH YOU.  YOU WILL TAKE TURNS WITH THE" :
         PRINT
   9015  PRINT "COMPUTER.  EACH OF YOU WILL BE TRYING TO"; :
         PRINT
   9020  PRINT "THINK OF NAMES OF PLACES SUCH THAT THE" :
         PRINT
   9025  PRINT "FIRST LETTER OF YOUR NAME IS THE SAME AS"; :
         PRINT
   9030  PRINT "THE LAST LETTER OF THE PREVIOUSLY USED" :
         PRINT
   9035  PRINT "PLACE NAME." : PRINT
*  9040  POKE 34,15
   9045  HOME : INPUT "ARE YOU READY? ";A$
   9065  IF LEFT$(A$,1) <  > "Y" THEN 9045
   9070  FOR I9 = 1 TO 1000 : NEXT I9
*  9080  TEXT : HOME
   9090  RETURN
```

Program 6-8. Play a Geography game.

....SUMMARY

String arrays are very convenient for maintaining a collection of string data in memory while our program is running. String arrays may be declared in a DIMension statement. Zero subscripts may be used if required.

We have seen in the example programs that it is easy to coordinate numeric values with string data by using a string array in tandem with a numeric array. Thus, the Kth element in the numeric array contains information about the string stored in the Kth element of the string array.

Problems for Section 6-3

1. In Program 6-8, which plays Geography, notice that the loop beginning at line 5ØØØ scans every name in the list. None of the names that have been added in this most recent game may be used by the computer, because they have all been used by the human player. Fix this so that the computer scans only those names which it "knows" at the start of the most recent game. (Suggestion: Establish a new variable, N2, which represents the number of names at the beginning of the current game.) Don't be tempted to change line 6Ø4Ø.
2. Modify the computer-response subroutine (Program 6-8g) so that the computer randomly selects a starting point in the names array. Be sure that if no name is found the computer scans from the beginning of the array to the random starting point.
3. Sometimes it is interesting simply to rearrange strings for display purposes. Write a program that enters the days of the week in a string array and displays them in the following format:

```
S   M   T   W   T   F   S
U   O   U   E   H   R   A
N   N   E   D   U   I   T
D   D   S   N   R   D   U
A   A   D   E   S   A   R
Y   Y   A   S   D   Y   D
        Y   D   A       A
            A   Y       Y
            Y
```

4. Write a program to enter a collection of names in a string array. Find the element that comes first alphabetically. Display it and its position in the array.

PROGRAMMER'S CORNER 6

Integer Variables in Applesoft

Generally we work with numeric values using conventional variables in Applesoft. This gives us up to nine decimal digits for calculation and

display. These numbers are referred to as real numbers. While one of the tremendous advantages of Applesoft is its real arithmetic, there may be times when we can solve our problem with integer arithmetic. This is especially significant when we are working with large arrays. Each of the numbers allocated in an integer array occupies two-fifths of the memory of each number allocated to a real array.

We set up conventional arrays by simply naming ordinary variable names. Applesoft distinguishes real and integer variables by requiring us to append a percent sign to indicate integer values.

```
100  DIM A%(100,100)
```

allocates 10201 integers in a 101-by-101 integer array. We can't even dimension such a real array on a 48K Apple. Simple variables may be established for integers in the same way.

```
100  B%=1.234
```

will result in storing the integer "1" in the integer variable B%.

.... Warning

Arithmetic using integer variables in Applesoft is not always the same as in Integer BASIC.

In Integer BASIC

```
-25/2  =  -12
```

However, in Applesoft

```
-25/2  =  -13
```

This is because Applesoft applies INT() function to any decimal values that we assign to integer variables. As with all things, it is important to get the whole story.

....A Word about Zero Subscripts and Space

If we are working on a program that requires arrays and we are having problems fitting into the available memory, we may be able to gain some space by using the 0 subscripts. Suppose we have a 100-by-100 array, because we really want 10000 elements. We may simply dimension the array with

```
100  DIM A%(99,99)
```

and subtract 1 from all subscript references in the program. This saves the memory required by 201 integer values or 402 bytes.

This effect increases as the number of dimensions in the array increases. Suppose we require an array to be 10 by 10 by 10. That comes to 1000 elements. If we dimension the array 10 by 10 by 10, we provide for 11 by 11 by 11, which is 1331 elements. That would be 331 more elements than the problem requires, a 33.1% excess.

Chapter 7

Using What We Know: Miscellaneous Applications

7-1...Looking at Integers One Digit at a Time

In general, the more detailed the control we have over a number in the computer, the more complex the problems we might expect to be able to handle. We also will find that, as we learn more about what goes on inside the computer, we will be able to apply more elegant solutions to problems. It is common to store a different piece of information in each digit of a number. It is also common to group digits in twos or threes for this purpose. Part numbers, serial numbers, and course numbers are just a few examples of this. In this section we will develop methods of breaking up numeric values into their separate digits.

.... Using MOD in Integer BASIC

One very convenient and fast method of getting at the digits of an integer is to use the MOD operator in Integer BASIC.

```
N MOD 1Ø
```

is always the units digit of any integer. Once we know what the units digit is, we can "throw it away" or "peel it off" by dividing the number by 1Ø. If the result after division by 1Ø is not Ø, then we must determine the next digit. We simply repeat this 2-step process until the value of N becomes Ø (zero). It looks like Program 7-1.

```
   100 INPUT "AN INTEGER",N
   200 D=N MOD 10
*  210 PRINT D,
   220 N=N/10
*  230 IF N<>0 THEN 200
   240 END
```

Program 7-1. Pick a number apart in Integer BASIC.

```
>RUN
AN INTEGER?32547

7        4        5        2        3
```

Figure 7-1. Execution of Program 7-1.

This method is clear and easy to code. You doubtless noted that the digits came out in reverse order. At line 210, we can easily save the digits in a five-element array. Then after line 230 we can display the digits in the proper order. This is left as an exercise.

.... Using Successive Division in Applesoft

Consider the number 2789. The 2 means 2000, which may be written $2 * 10^3$; the 7 means 700, which may be written $7 * 10^2$; the 8 means eight 10s, which may be written $8 * 10^1$; and the 9 means 9 units, which may be written $9 * 10^0$. Looking at the numbers step by step,

$$\begin{aligned} 2789 &= 2 * 10^3 + 789 \\ 789 &= 7 * 10^2 + 89 \\ 89 &= 8 * 10^1 + 9 \\ 9 &= 9 * 10^0 + 0 \end{aligned}$$

This is an example of the general relationship

$$N = I * 10\text{^}E + R$$

where I is the integer quotient found by

$$I = INT(N / 10\text{^}E)$$

and an iterative process whereby the new N is the old R and the value of E is decreased by 1 for each iteration. Solving for R we get

$$R = N - I * 10\text{^}E$$

For 9-digit integers the value of E will have to begin at 8 and go to 0 STEP −1. Carefully study Program 7-2.

```
   100  PRINT "INPUT AN INTEGER";
   110  INPUT N
   120  IF N = 0 THEN  END
   130  FOR E = 8 TO 0 STEP  - 1
*  140 T = 10 ^ E
   150 I =  INT (N / T)
   160  PRINT I;"  ";
   170 R = N - I * T
   180 N = R
   190  NEXT E
   200  PRINT : PRINT
   210  GOTO 100
```

Program 7-2. Access digits by successive division.

Note line 140. In that line we simply save the value of 10 ^ E. Exponentiation is a slow process, and there is no need to have the computer do it twice for each value of E.

```
]RUN
ENTER AN INTEGER?123456789
1  2  3  4  5  6  7  8  8

ENTER AN INTEGER?999
0  0  0  0  0  0  9  9  8

ENTER AN INTEGER?0
```

Figure 7-2. Execution of Program 7-2.

A quick look at the display (shown in Figure 7-2) of the execution of our seemingly simple program reveals that something is terribly wrong.

We have created a situation in which the computer is rounding things off internally in such a way that accuracy is lost. Even 999 comes out 998. If we insert a statement at line 175 to display the values for R, we will see that it is always just a little low. The easiest way to fix this is to calculate the value of R by rounding off to the nearest unit. This is left as an exercise.

.... Using STR$ in Applesoft

A very easy method for getting at the individual digits of a number is provided by the STR$ function in Applesoft. Once we get a number stored as a string, we can use the LEFT$, MID$, and RIGHT$ functions to pick numbers apart as we see fit. It becomes very easy to pick out any starting point and any number of digits. We can scan the number to look for a decimal. We can use the LEN function to find how many characters it takes to display the number. For demonstration purposes, let's write a little program to display each digit of a number individually.

```
100  PRINT "ENTER A NUMBER";
110  INPUT N
120  IF N = 0 THEN  END
130 A$ =  STR$ (N)
140  FOR I = 1 TO  LEN (A$)
150  PRINT  MID$ (A$,I,1);" ";
160  NEXT I
170  PRINT : PRINT
180  GOTO 100
```

Program 7-3. Using STR$ to separate numeric digits.

```
]RUN
ENTER A NUMBER?695.32147
6 9 5 . 3 2 1 4 7

ENTER A NUMBER?147896325523698741
1 . 4 7 8 9 6 3 2 6 E + 1 7

ENTER A NUMBER?0
```

Figure 7-3. Execution of Program 7-3.

Note the second number entered in Figure 7-3. Since it is represented in exponential notation for display purposes, that is the format used by STR$(). In the case of decimal numbers and exponential notation we will have to construct more logic to determine the actual numeric value represented by a particular digit according to its position in the number.

.... SUMMARY

We have seen several methods for picking apart numbers digit by digit in a computer. The MOD operator in Integer BASIC is convenient. In Applesoft either successive division or the STR$() function may be used. We discovered that we had to round off the value we got after removing the leftmost digit each time. Using the STR$() function we can easily access any individual digit in any order.

Problems for Section 7-1 .....

I 1. Rewrite Program 7-1, using a five-element array, so that the digits are displayed in the same order as they appear in the number as entered. Now we might want to think about a way to avoid having leading zeros displayed.

A 2. Rewrite line 170 of Program 7-2 so that the value in R is rounded to the nearest unit.

A 3. Modify Program 7-2 so that leading zeros are not displayed. Be careful that you don't eliminate all zeros!

I 4. The method of Program 7-2 can also be applied in Integer BASIC. In this case, we will not have the rounding-off errors that caused difficulty in Applesoft.

AI 5. Write a program to construct an integer by reversing the digits of an entered integer. Place the result in a numeric variable and PRINT its value.

AI 6. Find all 3-digit integers that are prime. Form new integers by reversing the digits and see if the new number also is prime. Print a number only if it and its reverse number are prime. There are 43 pairs of numbers, some of which appear twice.

AI 7. Do Problem 6, but eliminate duplicates.

7-2...Number Bases

The day-to-day world of business, commerce, and general communications reckons in the familiar base-10 number system. The ultimate reckoning of the computer is in base 2. Base 2 requires only the 2 digits "0" (zero) and "1" (one). Computers may represent a "1" with a positive voltage level or a magnetized state and a "0" by a 0 voltage level or a demagnetized state. Therefore, it is useful to be familiar with the base-2 number system. The base-2 number system is also referred to as the binary number system. A number is a number is a number is a number. The number does not change by virtue of being expressed in a different number system. As we change from one base to another, we may be using different symbols to name the same number. In the binary number system, there are only 2 possible digits.

Addition in base 2 is very simple. Either there is a "carry" as the result of two ones being added or there is not. Thus:

0 + 0 = 0
0 + 1 = 1
1 + 1 = 10

Multiplication is also simplified by the 2-possible-digit structure. When multiplying by "1" the digits shift according to the position of the "1", and when multiplying by "0" the result is "0". When multiplying by "1" in the rightmost position, the shift is 0. When multiplying by "1" in the 2nd position from the right, the shift is 1. If we choose to number the positions from right to left as 0, 1, 2, 3, . . . N, then the shift equals the position of the "1".

```
     1 * 101001 = 101001      (shift of 0)
    10 * 101001 = 1010010     (shift of 1)
```

and

```
   1000 * 101001 = 101001000  (shift of 3)
```

Thus:

```
   1Ø          11Ø11
 * 1Ø        *   1Ø1
 ----        -------
  1ØØ          11Ø11
              ØØØØØ
             11Ø11
            --------
            1ØØØØ111
```

Note that in the second multiplication example there is a carry across several positions.

One disadvantage of the binary number system is that it takes so many digits to represent numbers. For instance, 15 base 1Ø is written as 1111 in binary, and 127 base 1Ø is written 1111111 in binary. However, this is a disadvantage only to humans. Computers are not fazed by this. In fact, the computer is very good at accessing individual bits and turning them on or off 1 at a time. The number 255 base 1Ø is written 11111111 in binary. It requires 8 binary digits to represent the number 255. Each binary digit is referred to as a bit. Bits are collected into groups of 8 to form bytes. The Apple II and Apple II Plus are 8-bit machines—that is, they use electronic circuits in sets of 8 to represent numbers and instructions in memory. Everything the computer does is stored in a byte or a group of bytes. This is why a number of the limits for Apples are 255.

Each digit of any integer represents an integer power of the base. So the digits in binary represent

1, 2, 4, 8, 16, 32, 64, 128, 256, 512, etc., in base 1Ø,

corresponding to bit positions

Ø, 1, 2, 3, 4, 5, 6, 7, 8, 9, etc., in binary.

On Apples the largest true integer value allowed is 32767, and the smallest is −32767. That is 65535 numbers. Zero base 1Ø is Ø in binary, but 65535 base 1Ø is represented by 1111111111111111 in binary notation. That is 16 binary digits. We get 16 binary digits by grouping 2 bytes together. It takes 2 bytes to represent integers from Ø to 65535. In practice the leftmost binary bit is used to designate whether the integer stored in the other 15 bits is positive or negative. A "1" indicates negative, and a "Ø" indicates positive. Thus, for 2-byte storage, we are limited to the range of −32767 to +32767 as mentioned above.

.... Decimal to Binary

Let's get started by writing a program to convert decimal to binary. If the base-1Ø number we have is odd, then the first base-2 digit on the right is a "1". If we have an even base-1Ø number, then the 1st base-2 digit on the right is a "Ø". Now, to move the base-2 decimal point 1 to the left, we

divide our base-10 number by 2 and ignore the decimal part. We can ignore the decimal part by simply chopping it off. This process for eliminating the decimal part of a number is called *truncation*. If the truncated result is 0, then we are finished. If the truncated result is non-0, then we repeat the process for the next binary digit. Consider the process for 53:

	53	is odd	1
divide by 2 and truncate	26	is even	01
divide by 2 and truncate	13	is odd	101
divide by 2 and truncate	6	is even	0101
divide by 2 and truncate	3	is odd	10101
divide by 2 and truncate	1	is odd	110101
divide by 2 and truncate	0	we have finished and	
	53 base 10 = 110101 base 2		

Now we simply need to work out a way to print the results, and a program will be forthcoming. The method we use is to store the digits in a 16-element array as we determine them. We store the rightmost (or low-order) digit in the 16th element, the 2nd digit in the 15th element, and so forth until finished. Later this can easily be expanded to accommodate larger numbers.

.... Using MOD in Integer BASIC

How do we know if a number is odd or even? In Integer BASIC we can evaluate the remainder mod 2. Any number mod 2 is 0 if the number is even and 1 if the number is odd. While the odd-versus-even comparison does not apply to other bases, the remainder mod N is the correct digit for conversion to base N. Note that, of course, the Integer BASIC MOD function requires base-10 arguments and returns base-10 values. See Program 7-4.

```
100 REM * CONVERT DECIMAL TO BINARY
110 DIM A(16)
120 FOR J=1 TO 16
130 A(J)=0
140 NEXT J
200 INPUT "ENTER AN INTEGER",I
210 IF I<=0 THEN 999
296 REM
298 REM  * LOAD THE ARRAY
300 FOR J=16 TO 1 STEP -1
310 A(J)=I MOD 2
320 I=I/2
360 NEXT J
396 REM
398 REM  * DISPLAY RESULTS
400 FOR J=1 TO 16
410 PRINT A(J);" ";
```

```
420 NEXT J
455 PRINT : PRINT
460 GOTO 120
999 END
```

Program 7-4. Convert decimal to binary.

```
>RUN
ENTER AN INTEGER?127

0 0 0 0 0 0 0 0 1 1 1 1 1 1 1

ENTER AN INTEGER?32512

0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0

ENTER AN INTEGER?0
```

Figure 7-4. Execution of Program 7-4.

Note that Program 7-4 prints all of the leading zeros. We might want to eliminate them. We could now easily display the digits in adjacent positions.

.... Using Applesoft

We will store the digits in an array as described earlier. Since Applesoft doesn't have the MOD function, we will handle conversion to base 2 a little differently. For any base-10 number, if division by 2 comes out even, then the corresponding base-2 digit is 0. If division by 2 leaves a decimal portion, then the corresponding base-2 digit is 1. This we can easily do with 2 lines of Applesoft code:

```
310  IF I / 2 =  INT (I / 2) THEN A(J) = 0
320  IF I / 2 < > INT (I / 2) THEN A(J) = 1
```

Line 310 enters a 0 in the Jth element if the integer is divisible by 2 and line 320 enters a 1 in the Jth element if the integer is not divisible by 2. Examine Program 7-5.

```
100  REM  * CONVERT DECIMAL TO BINARY
110  DIM A(16)
200  INPUT "ENTER AN INTEGER? ";I
210  IF I <  = 0 THEN 999
220  IF I < 65536 THEN 300
230  PRINT "TOO LARGE" : PRINT : GOTO 200
296 :
298  REM  * LOAD THE ARRAY
300  FOR J = 16 TO 1 STEP  - 1
310  IF I / 2 =  INT (I / 2) THEN A(J) = 0
320  IF I / 2 < > INT (I / 2) THEN A(J) = 1
```

```
340 I =  INT (I / 2)
360  NEXT J
396 :
398  REM  * DISPLAY RESULTS
400  FOR J = 1 TO 16
410  PRINT A(J);" ";
420  NEXT J
455  PRINT : PRINT
460  GOTO 200
999  END
```

Program 7-5. Decimal to binary using successive division.

```
]RUN
ENTER AN INTEGER? 127
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1

ENTER AN INTEGER? 32512
0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0

ENTER AN INTEGER? 0
```

Figure 7-5. Execution of Program 7-5.

.... Binary to Hexadecimal

The hexadecimal number system reckons in base 16 because hex uses 16 possible digits. The hex digits are Ø, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. So 1Ø hex is 16 base 1Ø, and EF hex is 14*16 + 15*1 or 239 base 1Ø. Whereas the place values for binary representation are 1, 2, 4, and 8, the place values for hexadecimal representation are 1, 16, 256, and 4Ø96. (Note: In all numbering systems the place values are really 1, 1Ø, 1ØØ, and 1ØØØ, when expressed in the notation of the numbering system itself. 1, 1Ø, 1ØØ, and 1ØØØ in hex are written as 1, 16, 256, and 4Ø96 in base-1Ø notation.) It takes 4 binary digits to form a hex digit:

```
1Ø11  ØØØ1    binary
 B     1    = B1 hex
```

So, 2 hexadecimal digits may be used to represent any number stored in 1 byte, and 4 hexadecimal digits may represent 2 bytes. This is very convenient for use with an 8-bit machine.

The hexadecimal numbering system offers some advantages when working with a computer. B1 is more compact and much easier to read than 1Ø11ØØØ1. There are some parameters associated with computers that are just plain easier to remember in hex than in base 1Ø. Computer memory is often blocked off in segments containing 16384 bytes each. That is 16 times 1Ø24 or 4ØØØ bytes in hex. One common unit of measure for computer memory is the K. One K is 1Ø24 bytes. So, for a 64K machine, the four 16K segments begin at ØØØØH, 4ØØØH, 8ØØØH, and

CØØØH. Those numbers are much easier to remember than Ø, 16384, 32768, and 49152.

....Hexadecimal to Decimal

The conversion from decimal to hex is exactly analogous to the conversion from decimal to binary, except that we have to work out how to get the extra digits A through F into the picture. Since the extra-digit problem also occurs in the hex-to-decimal conversion, this is where we start.

Let's convert 1B3A hex to decimal:

The digit A	in the 1's	column represents	1Ø
The digit 3	in the 16's	column represents	48
The digit B	in the 256's	column represents	2816
The digit 1	in the 4Ø96's	column represents	4Ø96
1B3A	hex	equals	697Ø base 1Ø

To work in hex, our programs must have a way to accept hex input and to display hex output. Obviously this cannot be done with numeric variables. We may store the 16 hex digits in a string variable. All hex input should be checked to verify that no bogus digits have been entered. Let's start by writing an Applesoft program that simply requests hex input, verifies it, and displays the verified number.

```
  100  REM  * DEVELOP HEX INPUT/OUTPUT
  130 H$ = "0123456789ABCDEF"
  140  GOSUB 400 : REM  * REQUEST & VERIFY
  150  PRINT N$
  190  GOTO 140
  396 :
  398  REM  * REQUEST & CALL VERIFY
  400  PRINT : INPUT "HEX NUMBER? ";N$
  410 L =  LEN (N$)
  420  IF L = 0 THEN  END
  430  IF L < 5 THEN 440
  432  PRINT "TOO MANY DIGITS"
  434  GOTO 400
  440  GOSUB 700
  450  IF FL = 0 THEN 490
  460  PRINT "BAD FORMAT" : GOTO 400
  490  RETURN
  696 :
  698  REM  * VERIFY HEX STRING
  700 FL = 0 : REM  * GOOD INPUT
  710  FOR J = 1 TO L
  720  FOR K = 1 TO 16
*  730  IF  MID$ (H$,K,1) =  MID$ (N$,J,1) THEN 760
  740  NEXT K
```

```
750 FL = 1 : REM  * BAD INPUT
755  GOTO 790
760  NEXT J
790  RETURN
```

Program 7-6. Hex input/output.

```
]RUN

HEX NUMBER? ABCD
ABCD

HEX NUMBER? AFAF
AFAF

HEX NUMBER? HEX
BAD FORMAT

HEX NUMBER? FF
FF

HEX NUMBER?
```

Figure 7-6. Execution of Program 7-6.

Now, how do we get the computer to "know" that an "A" is 10 and a "B" is 11 and so on? Since the digits are not numeric, we have this problem even for "0", "1", etc., as well.

This is not so tough as it might seem at first. Line 730 of Program 7-6 gives us all the information we need. The value of K there tells us which digit in the sample string H$ matches the Jth digit of the input string. If K = 1 then the digit in H$ is a 0; if K = 16 then we come up with "F". So, subtracting 1 from K gives us the values from 0 to F corresponding to 0 to 15. Then, knowing which digit we are on tells us which "place" that digit represents. So we know what power of 16 to use.

The digit value is K − 1. The place is L − J. So the base-10 value is

(K − 1) * 16 ^ (L − J)

We simply need a numeric variable in which to accumulate this information. Using this information the subroutine at line 700 could easily return the base-10 value of the hex input. Simply set a numeric variable to 0 at about line 705 and accumulate at line 760, while moving NEXT J to line 770. This is left as an exercise.

.... SUMMARY

We have seen that the rationale for base 2 or binary is that the digits "0" and "1" can be represented as electrical states of one sort or another. The

hexadecimal number system is convenient because it correlates so nicely to data as it is stored in computer memory. Whereas it takes 8 digits to represent a byte of computer memory in binary, it requires only 2 hexadecimal digits. All conversion techniques rely upon determining the position of a particular digit and its actual value.

Problems for Section 7-2

AI 1. Write a program to convert binary to hex.
I 2. Modify Program 7-4 to eliminate leading zeros and display the result with no spaces.
A 3. Modify Program 7-5 to eliminate leading zeros and display the result with no spaces.
A 4. Modify Program 7-6 to do the conversion as described in this section.

7-3...Miscellaneous Problems for Computer Solution

We offer a few interesting problems for computer application here. Do not limit yourself to the problems suggested. You should be bringing your own problems to the computer. Although it is important to have problem suggestions in any book, you will find that a tremendous satisfaction comes with developing your own ideas on the computer.

.... Problems of General Interest

1. There is an old number puzzle about cows, pigs, and chickens that lends itself nicely to computer solution. A farmer has exactly $1Ø0 to spend on animals. He wants to buy at least 1 cow, at least 1 pig, and at least 1 chicken. Cows are $1Ø each, pigs are $3 each, and chickens are $.5Ø each. How many of each must he buy to have exactly 1ØØ animals?

At first, this looks like an easy algebra problem. One soon finds that we have only two equations with which to solve a problem having three unknowns. This is where the computer comes in. We simply try all combinations of cows, pigs, and chickens until these equations are satisfied:

$$1Ø * CO + 3 * PI + CH / 2 = 1ØØ$$
$$CO + PI + CH = 1ØØ$$

By observing that there must be many more chickens than either pigs or cows we could solve this by hand using trial and error. But

we still might become frustrated with the number of calculations required.

The key to this problem is to realize that each of the 3 numbers we are looking for must be an integer. We could easily write a program with 3 nested FOR . . . NEXT loops where CO goes from 1 to 1Ø, PI goes from 1 to 33, and CH goes from 2 to 1ØØ by 2s. If we do that, we will find that the program has to "think" for some time. We can greatly speed things up by using more of the information available to us. Clearly, if there must be at least 1 of each animal, there cannot be 1Ø cows or 33 pigs or 1ØØ chickens. There could be no more than 9 cows, no more than 29 pigs, and no more than 98 chickens. We can derive the greatest speed improvement by using the fact that once the number of pigs and cows to try has been established, we can find the number of chickens from

```
1ØØ - CO - PI
```

Next we check this number to see that it is even since the price is $Ø.5Ø. In Applesoft, we may test to see if CH divided by 2 is an integer. In Integer BASIC, we may use the MOD function for this. Write a program to solve this puzzle.

2. Sometimes it is fun to try to guess a number that someone else is thinking of. It is fairly easy to program a computer to play this simple game. Have the computer request the largest number from the user. Then the program should compute a random number in the range from one to the largest number. Next the program should ask for guesses from the user. Each guess should be checked. If the number is less than one or greater than the upper limit a message should put the user back on the right track. If the number is a correct guess, the program should say so. The program should also note whether the actual number is higher or lower than the most recent guess.

3. There are many famous chess puzzles that are appropriate for computer solution. A notable one is the eight-queens problem. In how many ways can eight queens be placed on a chessboard so that no queen attacks another?

This puzzle may be solved by using one eight-element array. Placing a queen in a position of the array assures that no two queens occupy the same row. A queen may be placed in the row by entering its column number there. Now we assure that no two queens occupy the same column by avoiding duplicate column numbers in the eight-element array. Finally we check for diagonal attack by noting that for two queens at positions (X,Y) and (X′,Y′), one diagonal is shared if $X-X'=Y-Y'$, while the other diagonal is shared if $X+X'=Y+Y'$. We need to have the computer test this for each queen in every column of one row. Write a program to print

the positions of all queens for each solution. *Note:* Your solution program may take a long time to produce results.

For more about the eight-queens problem see the October 1978 and February 1979 issues of *Byte* magazine.

4. It is always instructive to learn about the cost of homeownership. Aside from the ongoing costs of painting, fixing the roof, real estate taxes, and insurance, there is the ever-present mortgage interest. Most mortgages are set up so that the monthly payment stays constant. In the beginning, there is a large interest payment and a small payment toward the principal. At the end, the interest payment is small and more goes toward the principal. The following formula may be used to calculate the monthly payment:

$$PA = P \frac{I(1 + I)^N}{(1 + I)^N - 1}$$

where:

P is the principal
I is the monthly interest rate
N is the number of months

Write a program to request the principal, annual interest rate, and number of years. Have the program display the monthly payment, the total amount paid, and the total interest paid.

.... Math-Oriented Problems

1. Every positive integer may be expressed as the sum of the squares of four integers. Zero may be included as one or more of those integers to be squared. For example:

$$1 = 0^2 + 0^2 + 0^2 + 1^2$$

Write a program to find all sets of four such integers for a requested integer. Be careful about efficiency in this one. Test your solution with small integers before trying large ones!

2. Suppose you have to find the greatest common factor of 23902 and 15096. What would you do? The famous mathematician Euclid would have found the remainder after dividing 23902 by 15096, which is 8806. Then he would have found the remainder after dividing 15096 by 8806, which is 6290. Then he would have continued this pattern as follows:

```
23902 = 1 * 15096 + 8806
15096 = 1 *  8806 + 6290
 8806 = 1 *  6290 + 2516
 6290 = 1 *  2516 + 1258
 2516 = 2 *  1258 +    0
```

Next, Euclid would have reasoned that since the remainder of the last division was Ø, the greatest common factor must be the last divisor, in this case 1258. This method required only 5 interations. How many would it have taken using other methods?

3. The sieve of Eratosthenes is an ingenious method for generating prime integers. Write down all the integers from two to the desired upper limit. Now keep the first number and cross out all multiples of it. Now keep the next un-crossed-out number and cross out all multiples of it. Repeat this process until there are no more numbers to cross out. The remaining numbers are prime.

   There are two areas in this algorithm that are pitfalls for unnecessary extra processing. First, if the first multiple in any case has already been crossed out, then all other multiples will also have been crossed out. Second, we only have to check for uncrossed-out integers up to the square root of the largest number in the original range.

   This algorithm can easily be implemented in an array. First, enter the integers from two to the upper limit into the array elements two through the upper limit. Next, use FOR . . . NEXT to access the multiple positions in the array. Set the contents of any element to be crossed out to zero. Finally, PRINT all subscript positions for which the element is not zero.

4. A perfect number is an integer the sum of whose proper factors is the integer itself. The proper factors of 15 are 1, 3, and 5. The sum of the factors of 15 is 9. Therefore 15 is not a perfect number. The proper factors of 6 are 1, 2, and 3. The sum of the proper factors of 6 is 6. Thus 6 is called a perfect number. Write a program to find the first 4 perfect numbers. Since the 5th perfect number is 33,55Ø,336, and there is a significant amount of execution associated with determining "perfectness," we would be unwise to test each integer up to that one! Even the first 4 will take some time to find in BASIC. It turns out that there don't seem to be any odd perfect numbers, so let's test only even numbers.

5. Euclid was an active mathematician! He concluded that all possible even perfect numbers are of the form

   $$N = 2^{(E-1)} * F$$

   where

   $$F = 2^E - 1$$

   and F is an odd prime.

   Using Euclid's algorithm, write a program to calculate perfect numbers. Try a range of 2 to 15 for E.

6. Pythagorean triples are sets of 3 integers that can be the sides of a right triangle. Thus, the sum of the squares of the 2 smaller *inte-*

gers equals the square of the largest one. The first Pythagorean triple is 3, 4, 5. Write a program to generate Pythagorean triples.

7. The number π has fascinated mathematicians for many centuries. Values for π may be calculated in a variety of ways. The following sequence is known to approach the value of π:

$$4(1 - 1/3 + 1/5 - 1/7 + 1/9 - 1/11 \ . \ . \ .)$$

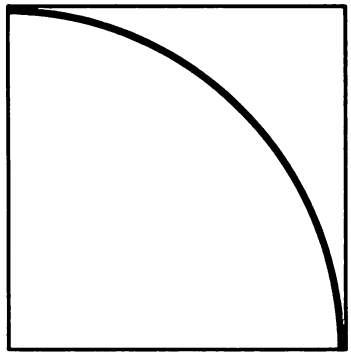
Write a program to evaluate this sequence for several large numbers of terms.

8. There are many sequences that approach π as the number of terms increases. The following is another one:

$$2 + 16\left[\frac{1}{1*3*5} + \frac{1}{5*7*9} + \frac{1}{9*11*13} + \cdots\right]$$

Write a program for this sequence. If you also did Problem 6, which sequence converges faster?

9. One method for approximating the value of π derives from the fact that the area of a circle is known to equal πR^2. A circle having a radius of 1 has an area of π. Thus, if we inscribe a circle in a square having a side of 2 and examine one quarter of the figure, we have a quarter circle inscribed in a square with side of 1.



The area of the square is 1 and the area of the quarter circle is $\pi/4$. If we have some way of measuring the area of the quarter circle, then we simply multiply that number by 4 to get an approximation of π.

If we generate random values between 0 and 1 for X and Y, we will always get a point in the square. Sometimes we will get a point in the circle. The ratio of the number of times the point falls in the circle to the number of points selected is proportional to the areas of the quarter circle and the square. If we get 80 points in the circle out of 100 points selected, then the approximation of π we come up

with is 4 * (8Ø/1ØØ) or 3.2. Write a program to calculate π in this way for 1ØØ, 5ØØ, 1ØØØ, and 5ØØØ random points. Assume that if a point lands on the circle then it counts as part of the circle.

You might experiment to see whether or not excluding points that fall on the circle has an impact on how fast the value you get approaches the known value of π (π = 3.1415926536 . . .).

PROGRAMMER'S CORNER 7

Writing a Program Menu.....

....Introduction

It is common practice where programs are run on a fast video display to present the user with a list of options. Usually the options are numbered and the user simply enters the number of the preferred option followed by a carriage return.

A sample menu might look like Figure 7-7.

```
1)  PLAY TIC TAC TOE

2)  SUPER LO-RES DEMO

3)  QUIT
```

Figure 7-7. Sample menu.

Note option 3. This is very important. By providing this option right in the menu we give the user proper control of the program. Many of the programs for sale in computer stores and by mail order are "menu-driven"—that is, they have a menu. The quality of menus is not consistent. One of the most common problems is the failure to include an option to terminate the program. We have to press CTRL-C or RESET to do that. Sometimes we even have to shut the machine off and back on again to run other programs. In some cases this is done to make it difficult for the user to make unauthorized copies of a program.

Another common affliction is that entering something not in the list of choices produces a messy display. In some cases the menu even begins to disappear from the screen if we enter several out-of-range choices. With some programs, pressing an extra key before the menu even appears on the screen produces surprising results. We will endeavor to write a menu program that avoids all of these problems.

.... Developing the Menu Routine

Each of the options in the menu may be a subroutine in a single program or each may be a separate program. That doesn't matter much. What we are about here is to develop a good menu-processing routine.

First of all, any graphics should be cleared out with the TEXT statement. Next, the text screen should be cleared with HOME or CALL −936.

Next, we should give some thought to how we take the choices from the keyboard. We may use an INPUT request or we can use PEEK(−16384) to process the keystrokes directly. In Applesoft we can use GET.

INPUT is quick and easy to code. However, if we code

```
940  INPUT X
```

and the user enters anything other than a numeric value, BASIC takes over, displaying error messages and rerequesting data. This results in a messy screen and may cause the menu to scroll out of sight. We could use

```
940  INPUT X$
```

and then convert the response to a numeric with the VAL function in Applesoft. This nicely handles the situation where the user fails to enter a numeric response. In any program where we may expect the user to know enough to press the RETURN key this is a good method to use.

PEEK(−16384) gives us the ultimate in control. We can request a character from the keyboard and not worry about whether or not the user knows enough to press the RETURN key. Programming with PEEK(−16384) will require a little more effort to write the BASIC routine. This seems worth doing. Once we have written a menu routine that works, we may plan future programs that can use it. All we have to change will be the names of the options and the control routine. Program 7-7 does it all.

```
90  DIM A$(10)
98  REM  * TEST THE MENU SUBROUTINES
100  GOSUB 9400
110  GOSUB 9000
120  IF S = N0 THEN  END
130  PRINT "YOU CHOSE - ";A$(S)
140  FOR I = 1 TO 1200 : NEXT I
150  GOTO 110
8994 :
8996  REM  * "DO THE MENU"
8998  REM  * RETURN SELECTION IN S
9000  TEXT : CALL  - 936
9002  VTAB 1 : HTAB 18 : PRINT "MENU"
9004  VTAB 3
9006  FOR I = 1 TO N0
9008  PRINT I;")  ";A$(I) : PRINT
9010  NEXT I
```

```
9020  PRINT "YOUR CHOICE: ";
9022  POKE  - 16368,0
9024 CH =  PEEK ( - 16384)
9026  IF CH < 128 THEN 9024
9028  POKE  - 16368,0
9030 CH = CH - 128 : PRINT  CHR$ (CH);
9032 S =  VAL ( CHR$ (CH))
9034  IF S >  = 1 AND S <  = N0 THEN 9088
9036  HTAB 19 : PRINT "NOT OFFERED"
9038  FOR I = 1 TO 1200 : NEXT I
9040  GOTO 9000
9088  PRINT
9090  RETURN
9394 :
9396  REM  * READ MENU OPTIONS
9398  REM  * NUMBER OF OPTIONS IN N0
9400  READ N0
9405  IF N0 < 10 THEN 9410
9407  PRINT "TOO MANY OPTIONS" : STOP
9410  FOR I = 1 TO N0
9420  READ A$(I)
9430  NEXT I
9490  RETURN
9495 :
9500  DATA  9
9510  DATA  PLAY TIC TAC TOE
9515  DATA  SUPER LOW-RES DEMO
9520  DATA  SWELL SOUNDS
9525  DATA  GOLF EXTRA
9530  DATA  COMPLICATED ARITHMETIC
9535  DATA  NEXT OPTION
9540  DATA  ANOTHER ONE
9545  DATA  THIS IS THE LAST OPTION
9555  DATA  QUIT
```

Program 7-7. Process a menu.

We have set up this menu program so that the options are entered in data statements. Lines 9000 to 9090 take care of displaying the menu and accepting a response from the keyboard. Note that line 9022 clears the keyboard before line 9024 reads it. In this way, no stray characters will be read in before the user is ready. Once an acceptable character has been found according to line 9026, we again clear the keyboard so the character found there is not read again later on in the program. Note in line 9030 that we subtract 128 from the code used for the character coming in from the keyboard. This is very important so that internal code matches a code expected by the VAL function. Remember that VAL returns 0 for any character other than the digits 0 through 9. It is usual to work with internal codes in the range of 0 to 127. We have even arranged to display *the* message "NOT OFFERED" on the same line as the question.

We might want more than 9 options. One method for handling this is to break the selections into 2 categories so that each category has 9 or fewer options. This is not a bad idea anyway. Simply include an earlier question that tells the program which category to offer. This structure can be extended to provide various "levels" of menu where some selections bring forth another menu offering another selection. Finally a selection takes the user into the process or program desired.

If you simply must have more than 9 options, then you might try using hex digits to get up to 15 options, or use the alphabet. Alternatively, it is very easy to use INPUT with a string variable and VAL to get any number of options. This is fine as long as it is permitted to expect the user to press the RETURN key.

Chapter 8

The Disk

8-1...What Is DOS?

DOS stands for "Disk-Operating System." The disk is contained in the little square envelope that we insert into the disk drive. The disk drive is the machinery required to read and write the disk itself. It is DOS that allows us to save programs on a disk. This single capability often justifies the purchase of a disk drive. With a disk system we may save many programs on one disk and retrieve them later using program names. Using DOS, we may request that the computer display the name of each program on a given disk for us. This is a tremendous improvement over using cassette tapes for saving programs. The disk is about 20 times as fast and much more reliable.

Furthermore, we can also easily use a disk to store data. The ability to store data on a disk turns our computer into a powerful data processing system. We may now use an Apple to handle a name-and-address list of hundreds or even thousands of names. We may enter statistical data with one program and use one or more other programs to perform a variety of analytic processes. It will not be necessary to enter the data separately for each program.

Data stored on a disk is referred to as a *data file*. Apple calls these files *text files*. Actually programs stored on disk are files also, but programs are recognized by BASIC as special.

DOS is really an extension to BASIC. There is a collection of BASIC keywords that enable us to utilize the disk. When we start up an Apple

with a disk this collection of disk-oriented keywords is incorporated into whichever BASIC we are using. We can even switch BASICs, and the DOS keywords remain for us to use. The command INT switches to Integer BASIC while FP switches to "Floating Point" or Applesoft BASIC. All this assumes that your Apple is appropriately equipped with the language you request. We can request FP even though we are already in Applesoft. Either of the commands wipes out any program currently in memory. So, be careful.

We can request a display of the contents of a disk with the DOS keyword CATALOG. The computer will respond by turning on the disk drive and displaying the name of each program and file as found on the directory. The disk directory is maintained by DOS. Along with each program name there is an "I", an "A", a "B", or a "T" (for Integer, Applesoft, Binary, or Text). A binary program is one that will execute on the Apple without the use of either BASIC. An entry labeled "TEXT" is a data file, which is the subject of this chapter. In addition, an asterisk may appear to indicate that a file has been LOCKed. A LOCKed file cannot be written to or deleted from the disk. When the display screen is filled DOS will wait until any character is typed at the keyboard before displaying an additional screen full. When the last entry has been displayed we will see either a (>) or a (]) indicating that BASIC is ready for the next command.

To save a program with the name "FIRST" on disk we simply type "SAVE FIRST". The disk will whirr for a few seconds. When it shuts off and the light goes out, typing "CATALOG" will reveal that indeed our program named "FIRST" is on the list. If it happens that we already have a program named "FIRST", it will be replaced by the new program. There is no way to retrieve the original version from this disk.

Once a program has been saved on a disk, two commands are available to us for using it. "LOAD FIRST" will transfer the program from the disk to the Apple memory. To execute the program we may next type "RUN". Or we might want to execute the program directly with the command "RUN FIRST".

If a program is no longer useful to us, it may be eliminated with the command "DELETE FIRST". It is best to assume that a deleted program cannot be recovered.

There is also a set of BASIC keywords that we may use to manipulate data on a disk. These may be used in programs to create files, write data, read data, and delete files. What these keywords do and how to use them are the primary content of this chapter.

8-2...What Is a File?

A file is simply some area of the disk where we may save data. As stated earlier, we may save programs on disk, too. When we save a program on

disk, DOS does everything for us. When we save data in a data file, it is up to us to do the organization of data.

One of the aspects about data files that encourages mystery is that they are invisible. Well, so are programs during execution. We have found that we could do fantastic things with programs even though we could see nothing going on until the final printed result. We will now expand our capabilities tremendously by using programs to create and access data files. We can LIST a program, but we are going to have to write programs to "LIST" any data file we create.

Data may be organized in a sequential format or arranged for random access. Some commands and techniques are the same for both sequential and random-access files. The most important thing to remember is that all control communications between any program and any file is done through the BASIC PRINT statement. That bears repeating: *All control communications between any program and any file is done through the* BASIC PRINT *statement.* The next important thing you must realize is that all file-control commands must be preceded by the character CTRL-D. Don't forget this. If we omit the CTRL-D character, then we simply see the contents of the PRINT statement displayed on the screen. *All file-control commands must be preceded by the character* CTRL-D. We will see in the next three sections exactly how to work with sequential and random-access files.

8-3...Sequential Files: An Introduction

Sequential files are easy to set up and use. We simply do everything beginning at the beginning. If we want to place 15 items in the file, then we simply write them in order from item 1 to 15. If we later wish to read the 14th item, then we read them all in order beginning with item 1 and stop when we get to item 14. Structurally, data in sequential data files is just like data in DATA statements of a program.

.... OPEN, WRITE, READ, and CLOSE

OPEN, WRITE, READ, and CLOSE are the four control commands required to perform any useful work with data files. Each must also name the file to be accessed. Each must be preceded by CTRL-D in a PRINT statement. Program 8-1 shows what it looks like.

```
90 D$ =  CHR$ (4)
100  PRINT D$;"OPEN TEST FILE"
110  PRINT D$;"WRITE TEST FILE"
```

Program 8-1. File-access routine.

Normally CTRL-D is invisible. Therefore it is very important to design our programs to make the "hidden" character come to our attention. In

Applesoft we can use a line like 9Ø above to achieve this. In Integer BASIC we should further document what we are doing:

```
9Ø D$="" : REM * CTRL-D
```

These precautions will save much confusion. Line 1ØØ in Program 8-1 contains the OPEN command. OPEN provides the communications link between our program and the file. If there is no file named "TEST FILE" when line 1ØØ is executed, then DOS will create it. This is done automatically for us.

Once a file is OPENed, we must tell DOS whether we wish to WRITE or READ. The actual writing and reading is done with PRINT and INPUT statements respectively. See Program 8-2.

```
9Ø D$ =  CHR$ (4)
1ØØ  PRINT D$;"OPEN TEST FILE"
11Ø  PRINT D$;"WRITE TEST FILE"
12Ø  PRINT "FIRST NAME"
```

Program 8-2. WRITE to a file.

In Program 8-2 line 9Ø makes the required CTRL-D visible. Line 1ØØ prepares the communications linkage between the program and the file for us, and line 11Ø sets up the file so that we may write data into it. Once the file is prepared for writing, the PRINT statement at line 12Ø will cause the characters of the string in quotes to be written to the file. Let's RUN it:

```
]RUN
■ <== the cursor stays here
```

Figure 8-1. Demonstrate faulty file access in Program 8-2.

An unexpected thing happens in Figure 8-1. Instead of displaying a bracket prompt (] or >), the cursor simply sits at the left position blinking away. That is because this program really contains an error. We have left an important part of file management to chance. We have done this here for you once so that you won't make this mistake numerous times before finding out what mysterious thing is destroying your files. Just as important as OPENing files is CLOSEing them. This program should contain the statement

```
13Ø  PRINT D$;"CLOSE TEST FILE"
```

This statement does all of the management associated with disconnecting our program from the file. If this is not done, we will encounter strange situations. It is not at all difficult to arrange things so that BASIC writes error messages out to the file! Our little demonstration program shows how to use the PRINT statement to write to a file. It is a simple matter

now to write a program that reads our one item from file TEST FILE. We use INPUT to read data from a data file. See Program 8-3.

```
90 D$ =  CHR$ (4)
100  PRINT D$;"OPEN TEST FILE"
110  PRINT D$;"READ TEST FILE"
120  INPUT A$
130  PRINT A$
140  PRINT D$;"CLOSE TEST FILE"
```

Program 8-3. READ data from a file.

Note that we want the printing at line 13Ø of Program 8-3 to go to the display screen. In order to do that we have a PRINT statement in a program that has an open file. But that is allowed here, because the file is opened for reading, not writing. However, if we have any file open for writing, then all printed results will go to that file. Even a message in a prompted INPUT statement will be written to a file open for writing.

We can make our programs more flexible by naming files using a string variable. Thus, our little demonstration program to read TEST FILE might be changed to look like Program 8-4.

```
80 F$ = "TEST FILE"
90 D$ =  CHR$ (4)
100  PRINT D$;"OPEN";F$
110  PRINT D$;"READ";F$
120  INPUT A$
130  PRINT A$
140  PRINT D$;"CLOSE";F$
```

Program 8-4. Demonstrate file name in a string variable.

By changing just line 8Ø we can easily use this program to read one string from any file named in F$.

Of course, we generally will store more than one string in a file. We may store hundreds of names in a file. Let's think about the logistics of building a file with a large number of names in it. If the file is truly to be very large, we have to consider that it may not fit on a single disk. As a practical matter, we will not be concerned with this for some time in our programming career. Writing the names into a file is no problem—we simply write a program that writes entry after entry. When we wish to read the data from the file the question comes up, "How many items are there?" We could use the ONERR GOTO statement and simply read until the file is OUT OF DATA, but there might be other sources of error in our program. It is much better to write the number of items in the file itself. We can simply make the first item in the file be the number of items to follow.

Remember the program we wrote to play Geography in Chapter 6? We wrote that program using subroutines so that it would be easy to convert to store the names in a disk file. This way, we can arrange to have the computer "remember" place names from one day to another. Let's first write a little program to store the four beginning names. See Program 8-5.

```
   10  REM  * INITIALIZE THE PLACES
        FILE FOR THE GAME OF GEOGRAP
        HY - WE ARE ENTERING THE FOU
        R LARGEST CITIES IN THE U. S
        . A.
   80 F$ = "PLACES"
*  90 N0 = 4
   100 D$ =  CHR$ (4)
   110  PRINT D$;"OPEN";F$
   120  PRINT D$;"WRITE";F$
*  130  PRINT N0
   140  PRINT "NEW YORK"
   150  PRINT "CHICAGO"
   160  PRINT "LOS ANGELES"
   170  PRINT "PHILADELPHIA"
   180  PRINT D$;"CLOSE";F$
```

Program 8-5. Write names to a file for Geography game.

When we run this program, we will have a file containing five items. The first item will be a "4" and the next four items will be four city names. It is important to note that the number we wrote to the file will be converted to the characters of the number, just like STR$. Thus, if N0 = 25, then there will be a "2" and a "5" in the file. However, we may retrieve the value with INPUT N0, just the same. Or in some special situation we might want to retrieve that number in a string variable. As the number representing how many names goes from one to two digits, the space required to store it in the file goes from one character to two. So when we rewrite the entire file, each of the place names will be located one character position further along in the file. We can program solutions to many problems without even realizing this. However, as we seek more elegant solutions, information of this kind will be important to have.

Let's now convert the Geography game from Chapter 6 to store names in a file. We simply need to replace the READ . . . DATA concept with a subroutine that reads the place names from the file into the array and provide a subroutine that writes out all of the names to the file at the end of this series of games.

The array version reads the names from DATA in a subroutine at line 8000. So we may simply replace that subroutine with a new one that reads the names from a file. See Program 8-6a.

```
7996 :
7998  REM  * READ NAMES FILE
8000  PRINT D$;"OPEN";F$
8005  PRINT D$;"READ";F$
8010  INPUT N0
8030  FOR I9 = 1 TO N0
8040  INPUT NA$(I9)
8050  NEXT I9
8060  PRINT D$;"CLOSE";F$
8090  RETURN
```

Program 8-6a. File-reading subroutine for Geography game.

Since there was no cleanup at the end of the array Geography game our new routine to write the names to the file at the end will be a new subroutine. Let's put it at 8500. Then the two file subroutines will be near each other in the final program. See Program 8-6b.

```
8496 :
8498  REM  * UPDATE NAMES FILE
8500  PRINT D$;"OPEN";F$
8510  PRINT D$;"WRITE";F$
8520  PRINT N0
8530  FOR I9 = 1 TO N0
8535  PRINT NA$(I9)
8540  NEXT I9
8580  PRINT D$;"CLOSE";F$
8590  RETURN
```

Program 8-6b. Write names to the file in the Geography game.

We can easily incorporate these two subroutines into the array Geography program. Next, we must provide the ever-necessary CTRL-D, assign the file name in F$, and modify the end-of-game logic to execute the subroutine at 8500 if the game just finished will be the last. All of this is provided by the five lines of Program 8-6c.

```
5 D$ =  CHR$ (4)
10 F$ = "PLACES"
90  IF LEFT$ (A$,1) = "N" THEN 140
140  GOSUB 8500 : REM  * REWRITE THE NAMES FILE
190  END
```

Program 8-6c. Changes in the control routine to convert array Geography to file Geography.

We present the complete program here for your convenience. See Program 8-6 on the next three pages.

```
*   5 D$ =  CHR$ (4)
*   10 F$ = "PLACES"
    20  DIM NA$(300),AV(300)
    30  GOSUB 8000 : REM  * READ NAMES ARRAY
    35  GOSUB 9000 : REM  * INSTRUCTIONS
    37  GOSUB 4000 : REM  * INITIALIZE AVAILABLE NAMES ARRAY
    40  GOSUB 7000 : REM  * COMPUTER STARTS
    50  GOSUB 6000 : REM  * PERSON RESPONDS
    58  IF PE$ = "QUIT" THEN 75
    60  GOSUB 5000 : REM  * RESPONSE OF COMPUTER
    65  IF CP$ <  > "QUIT" THEN 50
    75  PRINT "DO YOU WANT ANOTHER GAME";
    80  INPUT A$
    85  TEXT
*   90  IF LEFT$ (A$,1) = "N" THEN 140
    100  FOR I9 = 1 TO 1000 : NEXT I9
    120  GOTO 35
*   140  GOSUB 8500 : REM  * REWRITE THE NAMES FILE
*   190  END
    3996 :
    3998  REM  * INITIALIZE AVAILABLE NAMES ARRAY
    4000  FOR J9 = 1 TO N0
    4010 AV(J9) = 1
    4020  NEXT J9
    4090  RETURN
    4996 :
    4998  REM  * COMPUTER RESPOND
    5000  FOR I9 = 1 TO N0
    5010  IF LEFT$(NA$(I9),1) = RIGHT$(PE$,1) AND AV(I9) = 1
          THEN 5050
    5015  NEXT I9
    5020  PRINT : PRINT " I HAVE RUN OUT OF NAMES"
    5025 CP$ = "QUIT"
    5030  GOTO 5090
    5050 CP$ = NA$(I9) : AV(I9) = 0
    5060  PRINT "    I CHOOSE: ";CP$
    5090  RETURN
    5996 :
    5998  REM  * PERSON GO
    6000  PRINT
    6010  INPUT "   YOUR TURN: ";PE$
    6012  IF PE$ = "QUIT" THEN 6190
    6015  IF LEN (PE$) > 1 THEN 6030
    6020  PRINT "NAME TOO SHORT" : GOTO 6010
    6030  IF LEFT$(PE$,1) = RIGHT$(CP$,1) THEN 6040
    6035  PRINT "NO MATCH" : GOTO 6010
    6040  FOR I9 = 1 TO N0
    6045  IF PE$ = NA$(I9) THEN 6100
    6050  NEXT I9
    6055  IF N0 < 300 THEN 6065
    6060  PRINT "NO ROOM FOR MORE NAMES" : GOTO 6010
    6065 N0 = N0 + 1
    6070 NA$(N0) = PE$ : AV(N0) = 0
```

```
   6080  GOTO 6190
   6096 :
   6098  REM * "FOUND NAME"
   6100  IF AV(I9) = 1 THEN 6150
   6110  PRINT "USED ALREADY" : GOTO 6010
   6150 AV(I9) = 0
   6190  RETURN
   6996 :
   6998  REM  * COMPUTER BEGIN THE GAME
   7000  HOME
   7010 X9 =  INT ( RND (1) * N0 + 1)
   7020 CP$ = NA$(X9) : AV(X9) = 0
   7030  PRINT "FIRST PLACE : ";CP$
   7090  RETURN
*  7996 :
*  7998  REM  * READ NAMES FILE
*  8000  PRINT D$;"OPEN";F$
*  8005  PRINT D$;"READ";F$
*  8010  INPUT N0
*  8030  FOR I9 = 1 TO N0
*  8040  INPUT NA$(I9)
*  8050  NEXT I9
*  8060  PRINT D$;"CLOSE";F$
*  8090  RETURN
*  8496 :
*  8498  REM  * UPDATE NAMES FILE
*  8500  PRINT D$;"OPEN";F$
*  8510  PRINT D$;"WRITE";F$
*  8520  PRINT N0
*  8530  FOR I9 = 1 TO N0
*  8535  PRINT NA$(I9)
*  8540  NEXT I9
*  8580  PRINT D$;"CLOSE";F$
*  8590  RETURN
   8996 :
   8998  REM  * INSTRUCTIONS
   9000  TEXT : HOME
   9005  PRINT "THIS PROGRAM WILL PLAY A GEOGRAPHY GAME" :
         PRINT
   9010  PRINT "WITH YOU.  YOU WILL TAKE TURNS WITH THE" :
         PRINT
   9015  PRINT "COMPUTER.  EACH OF YOU WILL BE TRYING TO"; :
         PRINT
   9020  PRINT "THINK OF NAMES OF PLACES SUCH THAT THE" :
         PRINT
   9025  PRINT "FIRST LETTER OF YOUR NAME IS THE SAME AS"; :
         PRINT
   9030  PRINT "THE LAST LETTER OF THE PREVIOUSLY USED" :
         PRINT
   9035  PRINT "PLACE NAME." : PRINT
   9040  POKE 34,15
   9045  HOME : INPUT "ARE YOU READY? ";A$
   9065  IF LEFT$(A$,1) <  > "Y" THEN 9045
```

```
9070  FOR I9 = 1 TO 1000 : NEXT I9
9080  TEXT : HOME
9090  RETURN
```

Program 8-6. File-oriented Geography game.

Once again we have reaped tremendous benefits from good program organization and extensive use of subroutines. By segmenting the array Geography program we made it a relatively simple exercise to make the conversion to operate with a data file. We replaced one subroutine, added a subroutine, and made minor changes in the control routine. By making minor changes in a well-structured program we have made major changes in that program's behavior. It is important to realize that we have isolated all possible sources of error to small areas of the resulting program. If the first program had been badly put together, we would have found ourselves tinkering in numerous places to create the new program. The tinkered program would have contained far more potential error sources.

Problems for Section 8-3

1. Write a program that lists the place names in the Geography game file.
2. Try as people will, somebody will misspell a name in a game of Geography. Write a program that enables us to edit place names.
3. Write a program that will enable you to eliminate a place name from the Geography names file.
4. The Geography-game logic for the computer response scans the names array from item 1 every time. Modify the game so that the scan begins at some random point. Don't forget to come around to the beginning of the list after checking the last name.
5. The scan for the computer's turn in Geography covers the entire names array. That unnecessarily includes the names that have been added during the current game. Modify the program so that the scan for the computer's turn covers only those place names which came from the names file at the beginning of the current game.

8-4...More on Sequential Files

Let's explore sequential-files behavior in a little more detail. It is important to be familiar with the use of commas and carriage returns in sequential files. Generally, the carriage-return character is used to separate data items in sequential files. This character is automatically sent to the file by a PRINT statement with no trailing semicolon or comma. Suppose we

include the following statement to print data into a file:

```
140  PRINT "THIS, AFTER ALL IS THE MAIN POINT."
```

Some special things happen (or we wouldn't be doing this). The comma in the quoted string will be written into the file. If we go to read this with an INPUT statement in Applesoft, we will have to use a statement such as

```
140  INPUT A$,B$
```

We cannot read that data with two separate INPUT statements. INPUT statements used to read data from a file behave in the same manner as they behave reading data from the keyboard. On the other hand, in Integer BASIC, not only do we have to be sure that any string variables are dimensioned to accommodate enough characters, but we must read the string above containing a comma with a single string.

Contrast this with what happens when we use the following statement to write to a sequential file:

```
140  PRINT "ONE","TWO"
```

Even though the comma will separate the items in the display on our screen, it will be ignored in a file. The above statement will produce the same results as

```
140  PRINT "ONETWO"
```

or

```
140  PRINT "ONE";"TWO"
```

It should be clear that the reading of data from a sequential file must be carefully coordinated with the writing.

For most purposes, it is best to PRINT data items into the file one at a time. Let DOS provide the carriage return.

If you cannot avoid placing a comma in a file, then you can fool the system by enclosing the data item within quotes. This can be done in Applesoft with a statement such as

```
140  PRINT Q$;"A, B, AND C";Q$
```

where you have set Q$ = CHR$(34) earlier in the program. However, the quotes are sent to the file, and you will have to worry about that later when you READ the data.

Have you tried to print a quote in Integer BASIC? Does the above discussion give you an idea? It should. Just write a little program in Applesoft that creates a file with a quote in it. Then read that quote in an Integer BASIC program. If you don't have Applesoft, there is still a way to get a quote into a file. Whenever a file is open, all output that otherwise goes to the screen goes to the open file. So, executing a LIST command with a file opened for writing will create a text file containing our pro-

gram. If we make the program something simple like

```
1 PRINT ""
```

We know that the 13th and 14th characters are quotes. This means that we can write a little program that will place quotes in a data file. See Program 8-7.

```
 1 PRINT ""
10 D$="" : REM  * CTRL-D
20 PRINT D$;"OPEN QUOTE"
30 PRINT D$;"WRITE QUOTE"
40 LIST 1,1
50 PRINT D$;"CLOSE QUOTE"
60 END
```

Program 8-7. Listing a program to a file.

We can then READ that string into a string variable, confident that the quotes will be in the string. Then a statement such as

```
Q$=A$(13,13)
```

will provide us with a string that contains only a quote. Next we create a file and WRITE only Q$. Now we have a file that contains only a quote for future use in other programs.

The same procedure may be used to save a program as a data file. If we first change the screen width with POKE 33,33 then the extra spaces inserted by the LIST command will be eliminated. Once a program has been saved in this way it may be retrieved with a brand-new special command.

```
EXEC file name
```

EXEC file name will command BASIC to read the file from disk just as though it were being typed from the keyboard and embed the program statements into any program already in memory. This makes it possible to save numerous subroutines on disk and use them in many programs without the need to retype them for each new use. We may even save an Integer BASIC program and EXEC it into Applesoft. Of course only those statements which will work in Applesoft will be useful. We can change programs from Applesoft to Integer BASIC in the same manner.

8-5...Random-Access Files

Since entries in a file may vary in length they may occupy varying amounts of space. Therefore there is no way of predicting just where the 5th or the 50th entry might begin. So, for sequential files we must always read from the beginning of the file. When we write to such a file, the safest way is to write or rewrite the entire file. As the file becomes larger and *larger this* all takes more and more time.

All that changes with random-access files. This new file structure makes it possible to read the 25th entry, make changes, and rewrite it to the file without any risk of damaging any of the other entries. This is done by allocating a fixed amount of space for each entry. This means that in many applications there is some unused space in the file.

We use random-access files for all kinds of record keeping. The ability to access any data entry at will is ideal for applications where we will not be processing every entry every time we access the file. Contrast this with the Geography game, in which we clearly processed every entry in the file with every use of the program. Random-access files are used for name-and-address mailing lists, every conceivable financial accounting function, and stock-portfolio management. Recipes, home-management data, and magazine-article reference material are all appropriate for random-access files.

In many applications several files are linked together to form a system of files. An order entry might "point" off to a mailing-list file and an inventory file.

With sequential files the fundamental unit of storage is the character or byte. With random-access files the fundamental unit of storage is the record. A record is simply a collection of bytes. If 2Ø bytes are enough for the entries we plan to store, then we may organize our file into records that contain just 2Ø bytes. The record size is entirely up to us. We decide record size according to our application. It is important to study each application thoroughly and plan effectively how we will organize files to manage the data required. It is devastating to lay out a file structure with records holding 3 strings in 4Ø bytes only to find out after 3 long programs have been written that we should have 4 strings in 48 bytes.

Often a group of programs will be used to handle a file or system of files—one program to enter and delete entries, another to edit entries, and perhaps a third to print a nicely formatted report to display all of the data in the file.

The OPEN, READ, and WRITE statements used for random-access files are simply extensions of the corresponding statements for sequential access. OPEN carries an "L" value to set the length of each record in bytes. READ and WRITE carry an "R" value that specifies which record in the file to access. We'll examine these more closely later.

Let's develop a computerized name-and-address list. This is a common need for business and personal use. The idea here is to store all the names and addresses in a disk file. Then we may extract those we need for any particular situation. Names may be classified by a code. We might set up a personal family mailing-list file using H, W, or C to designate friends of husband, wife, or children. A business might use B and S for billing and shipping addresses.

In business it is common practice to arrange these names alphabetically or by zip code or by business volume. In order to achieve this we

would not rearrange the names file itself, but instead we would create a file that contains just a list of the records in the desired order. We might maintain several such lists of record numbers. Then we can easily write a program that will read a list of record numbers to print the corresponding name-and-address data from the data file in the desired order.

Let's organize a program to build the mailing-list data file. There are a number of major tasks involved. One part of the program needs to request all of the necessary data from the keyboard. Another will write the entry into the file. Another will have to determine where the new entry belongs.

We will have to organize the entry itself. We must decide what information belongs in an entry and how many characters to allow for each item and then calculate the necessary record size. Our program must include code to manage all these things. We need a routine that will write the entry in the data file. Probably the most important part of writing the program is deciding how to organize entries within the file.

When we sit down to enter the first name and address we know that the file is empty. After that we have no idea how many names are in the file. Therefore we have no idea where the next entry should go in the file. We could keep track of how many names there are on a piece of paper. Then we might just as well keep the names on paper too. The whole idea is to let the computer do the work. We need to develop a plan for keeping track of where things are. One scheme is to assign each entry its record number as an identification number and include that number as part of the data entry. The first name in the system is number 1, the second is number 2, etc. Now we can have the next number to be assigned saved in the file itself. A good place to do this is in record Ø. Lucky for us the record count begins at Ø. So, a file with no names in it should have a 1 stored in record Ø. We can easily write a little initialization program to do this.

Then after each new name is entered the program adds 1 to that value in record Ø. Next, we should be thinking about how we delete a name from a file even though we are preparing to write the program to place new entries in the file.

Deleting names from a mailing list can be handled in one of several ways. We could replace the name with the word "DELETED". Or we could develop a concept that provides that the most recently deleted entry record becomes immediately available for use by the next new entry. We can make deleted records available for new entries by setting up an available-space catalog within the file itself. To do this we include as an item of data, with the name and address, the record number itself. Then when an entry is deleted we store the number of the last deleted record in the deleted record and then store the number of the currently deleted record in record Ø along with the number of the next highest record in the file. This will leave a trail of deleted record numbers beginning with the number stored in record Ø. Now we have two numbers stored in record Ø:

the next record at the end of the file and the most recently deleted record. When we start up a new file, the most recently deleted record will be Ø.

This scheme provides a method for determining whether an entry has been deleted or not. Read the record. If the identification number equals the record number, then it is real data. If not, then the entry has been deleted, and the number is the record number of the previously deleted record. As an example of a file with some deleted records see Figure 8-2.

Ø	9 {on the end}, 8 {last deleted entry}
1	1 JONES JOHN . . .
2	2 SMITH WILLIAM . . .
3	3 HAYES MARY . . .
4	6 {deleted entry} . . .
5	5 BRADSHAW ELEANORE . . .
6	Ø {deleted entry (first one)} . . .
7	7 HOUGH HUGH . . .
8	4 {deleted entry} . . .
9	{never used}

Figure 8-2. Layout of used and deleted records.

Let's trace the available-space catalog in Figure 8-2. The 2nd number in record Ø is 8. Look at record 8. There we find a 4. Look at record 4. There we find a 6. Look at record 6. There we find a Ø. Thus the deleted records are 8, 4, and 6. When we use record 6 for a new entry, the program should place a Ø in record Ø where the 8 is now.

The entry program will have to look at the 2 record numbers stored in record Ø and decide whether to place the new entry at the end of the file or on a record from which a name has been deleted. That is easy. If the deleted record number is Ø the new name goes on the end. Otherwise use the deleted record.

It is important to observe in all this that even though we are de*sign*ing the program to enter data, it is necessary to think through the deleting

process thoroughly. We must design the whole system before actually coding any part of it.

We have entering and deleting pretty well under control. Now how about changing an entry? As long as each name has an identification number we can easily read the corresponding record and display each item as it appears, giving ourselves the opportunity to make changes in each case. We will need periodically to print up a list of the names with the IDs. It should be relatively easy to write a program to scan the file from beginning to end, displaying the data in each undeleted record. That program can easily select various categories according to the code stored in the code item.

We seem to have thought through four functions of our mailing-list system: new, delete, change, and display. We have mentioned the need to initialize the data file once to prepare it for entering data. Let's do that first. See Program 8-8.

```
90  REM  * INITIALIZE MAILING LIST FILE
95 D$=CHR$(4)
100  PRINT D$;"OPEN FIRST FILE"
110  PRINT D$;"WRITE FIRST FILE"
120  PRINT 1
122  PRINT 0
130  PRINT D$;"CLOSE FIRST FILE"
```

Program 8-8. Initialize mailing-list file.

Once this program has been run we may count on record 0 containing a 1 and a 0. Of course, we must assure that this program is never run again. The job of reconstructing such a file is better left to other people.

Let's now design the layout for a data record. See Table 8-1.

DATA ITEM	LABEL	MAXIMUM # OF CHARACTERS
Identification #	ID #	4 + 1
Code	CODE	5 + 1
Last name	LAST	20 + 1
First name	FRST	20 + 1
Address	ADDR	30 + 1
City	CITY	16 + 1
State	STAT	2 + 1
Zip	ZIP	5 + 1
Telephone	PHON	17 + 1
		119 + 9 = 128

Table 8-1. Record layout for mailing list file.

In Table 8-1 we have allowed 1 character for a carriage return at the end of each item in the entry. Note the large value for telephone. That

allows for the area code, an X, and a 4-digit extension. The total comes to 128 characters.

If we are careful about listing all of the above considerations we will have the structure of the control routine for our name-and-address-entry program. Once we have the control routine we may concentrate on a single subroutine at a time. See the list of functions in Figure 8-3.

1. Read data labels.
2. Read available space parameters.
3. Display next available ID and request data.
4. Terminate on null LAST name.
5. Prepare available space.
6. Write new entry.
7. Write available space info back to record zero.
8. Do it again.

Figure 8-3. List of functions for name-and-address-entry program.

Six of eight tasks listed in Figure 8-3 are appropriate for subroutines. Some of those subroutines will also be used by the other programs that we will be writing for our name-and-address system. For number four to terminate on null LAST name we need to provide a way for the data-requesting routine to send back a signal to quit. Number eight will simply direct the program to repeat the functions again beginning with number three.

We may arbitrarily select line numbers for the subroutines and for the control routine itself, and we will have our program substantially completed. See Program 8-9a.

```
200  GOSUB 1000 : REM  * READ DATA LABELS
210  GOSUB 900 : REM  * READ AVAILABLE SPACE PARAMETERS
220  GOSUB 800 : REM  * DISPLAY NEXT AVAILABLE ID AND
     REQUEST DATA
230  IF E1 = 1 THEN  END : REM  * TERMINATE ON NULL
     LAST NAME
240  GOSUB 700 : REM  * PREPARE AVAILABLE SPACE
250  GOSUB 600 : REM  * WRITE NEW ENTRY
260  GOSUB 500 : REM  * WRITE AVAILABLE SPACE INFO BACK
     TO RECORD ZERO
270  GOTO 220 : REM  * DO IT AGAIN
```

Program 8-9a. Control routine for mailing-list program.

We have six subroutines and two control statements in our main routine of Program 8-9a. Line 230 requires that the value of E1 be set to 1 if the operator desires to exit and set to any other value for any entry that is to be placed in the file. Line 270 simply uses a GOTO to repeat the request for another new entry. We will now write the subroutines one at a time.

We read the data labels at 1ØØØ. If we give some more thought to how to design the routine to take data from the keyboard, we should be able to come up with a creative scheme. We could surely ask the eight questions in eight statements using INPUT with prompt. For each of the eight inputs we could have a statement that checks to see if the entry is too long. Any changes in the file design will require changing that routine. Wouldn't it be a good idea to put the prompt labels and the maximum field sizes in DATA and read them into two arrays? Then major changes in the program can be made with simple changes in the DATA statements. Our DATA statements will come directly from the labels and character limits in Table 8-1. We can read the DATA into arrays with a FOR . . . NEXT loop. See Program 8-9b.

```
998  REM  * READ DATA LABELS AND LIMITS
1ØØØ  READ NØ
1Ø1Ø  FOR X9 = 1 TO NØ
1Ø2Ø  READ LA$(X9),LE(X9)
1Ø3Ø  NEXT X9
1Ø9Ø  RETURN
1996 :
1998  REM  * DATA LABEL & LIMITS
2ØØØ  DATA   9
2ØØ5  DATA   ID #,     4
2Ø1Ø  DATA   CODE,     5
2Ø15  DATA   LAST,    2Ø
2Ø2Ø  DATA   FRST,    2Ø
2Ø25  DATA   ADDR,    3Ø
2Ø3Ø  DATA   CITY,    16
2Ø35  DATA   STAT,     2
2Ø4Ø  DATA   "ZIP ",   5
2Ø45  DATA   PHON,    17
```

Program 8-9b. Read the data labels for mailing-list program.

In Program 8-9b NØ is the number of data items in an entry. The labels are stored in the LA$ array, and the maximum numbers of characters are stored in the LE array. The completed program should include an appropriate dimension statement.

The subroutine to read the available-space parameters is very simple. It just reverses the action of the initialization program. We need to select variables for the two available-space values. See Program 8-9c.

```
898  REM  * READ AVAILABLE SPACE
9ØØ  PRINT D$;"OPEN";F$
91Ø  PRINT D$;"READ";F$
92Ø  INPUT NS
93Ø  INPUT DS
94Ø  PRINT D$;"CLOSE";F$
99Ø  RETURN
```

Program 8-9c. Read available space in mailing-list program.

In Program 8-9c we have chosen to carry the new space in the variable NS and the deleted-space value in DS. We must note here that the completed program must save CTRL-D in the variable D$.

Now it is time to display the next available ID and request data. We said we would do this at 8ØØ. Since we have planned carefully, this will be very straightforward. The first job here is to determine the next actual available space. We choose to first make it new space. Then if there is any deleted space we reassign DS to the ID. We handle the label display and the data request with a FOR . . . NEXT loop. See Program 8-9d.

```
   798  REM  * PROCESS DATA ENTRY FROM KEYBOARD
   8ØØ ID = NS : IF DS <  > Ø THEN ID = DS
   8Ø3  PRINT
   8Ø5  PRINT LA$(1);": ";ID
   81Ø DA$(1) =  STR$ (ID)
   815  FOR I9 = 2 TO NØ
   82Ø  PRINT LA$(I9);"? ";
   825  INPUT DA$(I9)
*  83Ø  IF I9 = 3 THEN  IF  LEN (DA$(3)) = Ø THEN E1 = 1 :
        GOTO 89Ø
   835  IF  LEN (DA$(I9)) <  = LE(I9) THEN 845
   84Ø  PRINT "TOO LONG" : PRINT "    : "; : GOTO 825
   845  NEXT I9
   85Ø E1 = Ø
   89Ø  RETURN
```

Program 8-9d. Handle keyboard data entry for mailing-list program.

Note that in line 83Ø we set E1 to 1 if the response to the request for LAST name is of zero length. This will be the length if the program user simply presses the RETURN key without any preceding characters. We must include the DA$() array in the DIMension statement in the completed program.

Next we must prepare available space. What we do here depends on whether we are going to replace a deleted entry or write a new record. If we are going to use a new record we simply add one to the new-space variable and RETURN. If we are going to write this data to a previously deleted record then we must retrieve the record number that was written there when the deletion occurred. That number is essential for accurately maintaining the available-space catalog. Remember this from Figure 8-2?

Now we must pay attention to the size of the data entry as it relates to the file itself. When we OPEN a random-access file we must tell BASIC the record size. This is done in the OPEN statement with the ",L" option. The statement

```
100  PRINT D$;"OPEN FIRST FILE ,L128"
```

will perform the OPEN function and set the record length to 128 bytes as desired. We may place the record length in a numeric variable and the file name in a string variable and use a statement such as the following:

```
200  PRINT D$;"OPEN";F$;",L";LØ
```

The use of READ and WRITE with random-access files requires an "R" value to specify the record to read from or write to.

```
PRINT D$;"WRITE";F$;",R";R1
```

will prepare the file F$ so that the next PRINT statement goes to record R1.

```
698  REM * IF THIS ENTRY REPLACES DELETED DATA MAKE
     PREPARATIONS
700  IF DS = Ø THEN 76Ø
*  71Ø  PRINT D$;"OPEN";F$;",L";LØ
*  72Ø  PRINT D$;"READ";F$;",R";DS
730  INPUT DS
740  PRINT D$;"CLOSE";F$
750  GOTO 79Ø
76Ø NS = NS + 1
79Ø  RETURN
```

Program 8-9e. Prepare available space for mailing-list program.

We have used the ",L" and ",R" options in lines 71Ø and 72Ø of Program 8-9e. Note that in this subroutine either new space changes or deleted space changes but never both.

Once the available-space situation is taken care of we may actually write the entry to the file. This is a very short subroutine with no particular complications.

```
598  REM  * WRITE ENTRY
600  PRINT D$;"OPEN";F$;",L";LØ
610  PRINT D$;"WRITE";F$;",R";ID
620  FOR I9 = 1 TO NØ
630  PRINT DA$(I9)
640  NEXT I9
650  PRINT D$;"CLOSE";F$
690  RETURN
```

Program 8-9f. Write data entry in the mailing-list program.

And last but by no means least we must provide the subroutine that writes the available-space parameters to record Ø. This is exactly like the initialization program except that we must write NS and DS. See Program 8-9g.

```
498  REM  * WRITE AVAILABLE SPACE DATA
500  PRINT D$;"OPEN";F$
510  PRINT D$;"WRITE";F$
520  PRINT NS
525  PRINT DS
530  PRINT D$;"CLOSE";F$
590  RETURN
```

Program 8-9g. Write available-space parameters in mailing-list program.

Finally, in order for all of this to happen we must include CTRL-D in D$, 128 in LØ, the file name in F$, and the appropriate dimensioning statement.

```
9  REM  *  ID => ENTRY IDENTIFICATION NUMBER
1Ø  REM *  NS => NEW SPACE
11  REM *  DS => DELETED SPACE
3Ø D$ =  CHR$(4)
5Ø LØ = 128
6Ø F$ = " FIRST FILE"
7Ø DIM LA$(9),LE(9),DA$(9)
```

Program 8-9h. Program parameters for mailing-list program.

In Program 8-9h line 3Ø stores CTRL-D in D$ so that the file-access commands in the PRINT statements are visible. It is interesting to note that we can do a preliminary test of our program without writing to any files by eliminating line 3Ø. With D$ as the null string all file access will be converted to keyboard and text-screen access. Line 5Ø stores the file record length in the variable LØ. This makes it very easy to change the record size for another mailing-list application. Line 6Ø assigns the file name to F$. Again, this makes it easy to change the program to work with another name-and-address file.

```
9  REM  *  ID => ENTRY IDENTIFICATION NUMBER
1Ø  REM *  NS => NEW SPACE
11  REM *  DS => DELETED SPACE
3Ø D$ =  CHR$(4)
5Ø LØ = 128
6Ø F$ = " FIRST FILE"
7Ø DIM LA$(9),LE(9),DA$(9)
196 :
198  REM  * ENTER NAMES AND ADDRESSES IN A MAILING LIST
     FILE
2ØØ  GOSUB 1ØØØ : REM  * READ DATA LABELS
21Ø  GOSUB 9ØØ : REM  * READ AVAILABLE SPACE PARAMETERS
22Ø  GOSUB 8ØØ : REM  * DISPLAY NEXT AVAILABLE ID AND
     REQUEST DATA
23Ø  IF E1 = 1 THEN  END : REM  * TERMINATE ON NULL LAST
     NAME
24Ø  GOSUB 7ØØ : REM  * PREPARE AVAILABLE SPACE
25Ø  GOSUB 6ØØ : REM  * WRITE NEW ENTRY
26Ø  GOSUB 5ØØ : REM  * WRITE AVAILABLE SPACE INFO BACK
     TO RECORD ZERO
27Ø  GOTO 22Ø : REM  * DO IT AGAIN
496 :
498  REM  * WRITE AVAILABLE SPACE DATA
5ØØ  PRINT D$;"OPEN";F$
51Ø  PRINT D$;"WRITE";F$
52Ø  PRINT NS
525  PRINT DS
53Ø  PRINT D$;"CLOSE";F$
59Ø  RETURN
```

```
596 :
598  REM  * WRITE ENTRY
600  PRINT D$;"OPEN";F$;",L";LØ
610  PRINT D$;"WRITE";F$;",R";ID
620  FOR I9 = 1 TO NØ
630  PRINT DA$(I9)
640  NEXT I9
650  PRINT D$;"CLOSE";F$
690  RETURN
696 :
698  REM * IF THIS ENTRY REPLACES DELETED DATA MAKE
     PREPARATIONS
700  IF DS = Ø THEN 76Ø
710  PRINT D$;"OPEN";F$;",L";LØ
720  PRINT D$;"READ";F$;",R";DS
730  INPUT DS
740  PRINT D$;"CLOSE";F$
750  GOTO 79Ø
760 NS = NS + 1
790  RETURN
796 :
798  REM  * PROCESS DATA ENTRY FROM KEYBOARD
800 ID = NS : IF DS <  > Ø THEN ID = DS
803  PRINT
805  PRINT LA$(1);": ";ID
810 DA$(1) =  STR$ (ID)
815  FOR I9 = 2 TO NØ
820  PRINT LA$(I9);"? ";
825  INPUT DA$(I9)
830  IF I9 = 3 THEN  IF  LEN(DA$(3)) = Ø THEN E1 = 1 : GOTO
     89Ø
835  IF  LEN (DA$(I9)) <  = LE(I9) THEN 845
840  PRINT "TOO LONG" : PRINT "    : "; : GOTO 825
845  NEXT I9
850 E1 = Ø
890  RETURN
896 :
898  REM  * READ AVAILABLE SPACE
900  PRINT D$;"OPEN";F$
910  PRINT D$;"READ";F$
920  INPUT NS
930  INPUT DS
940  PRINT D$;"CLOSE";F$
990  RETURN
996 :
998  REM  * READ DATA LABELS AND LIMITS
1000  READ NØ
1010  FOR X9 = 1 TO NØ
1020  READ LA$(X9),LE(X9)
1030  NEXT X9
1090  RETURN
1996 :
1998  REM  * DATA LABEL & LIMITS
```

```
2000  DATA   9
2005  DATA   ID #,     4
2010  DATA   CODE,     5
2015  DATA   LAST,    20
2020  DATA   FRST,    20
2025  DATA   ADDR,    30
2030  DATA   CITY,    16
2035  DATA   STAT,     2
2040  DATA   "ZIP ",   5
2045  DATA   PHON,    17
```

Program 8-9. Entering names in a mailing-list file.

This program is intended to be a simple example of a workable mailing-list data entry program. Using the preceding discussion and some of the routines of this program you should be able to develop programs to delete entries, change entries, and print mailing labels.

There are many areas in which this program can be made more flexible. We might change the design a little to place the record size in record 0 of our file along with the available-space information already there. We might request the mailing-list file name from the program operator. We might eliminate the DATA statements from the program by placing that data in a companion file. The benefits of doing things this way are tremendous. With all of the information about the mailing list stored in a file, our one program can be used to process many different mailing lists. We can handle different numbers of items in a record, we can handle different sets of item size limits, we can handle different record sizes, and we can handle different labels, all in the same program. We will soon find that we have to write a program to manage the companion file that contains all of this useful information. That is a small price to pay. When we can change the behavior of a program by changing data in a file, we approach data-base-management capabilities.

Programming for the delete and change functions can be handled either by writing separate programs or by including the new subroutines necessary right in Program 8-9. We could provide a menu that lets the user select which function is desired.

.... SUMMARY

Once we organize files in records of a fixed size we may get at any data entry in the file as long as we know where it is. This constitutes a tremendous advantage over sequential files. We can read the 200th entry just as quickly as we can read the first. In order to implement random-access files we must specify the record length and the record number.

The record size is specified in an OPEN statement with the ",L" option. The statement

```
910  PRINT D$;"OPEN DOOR ,L192"
```

will open a file named DOOR with a record length of 192 bytes. Following this statement we may prepare the file for reading at record 103 with the following statement:

```
920  PRINT D$;"READ DOOR ,R103"
```

Both the ",L" and ",R" options may be coded with the values in variables.

```
1000 X1 = 192 : Y1 = 103 : F$ = "DOOR"
1010  PRINT D$;"OPEN";F$;",L";X1
1020  PRINT D$;"READ";F$;",R";Y1
```

Lines 1000–1020 will perform in exactly the same way as lines 910 and 920 above.

Problems for Section 8-5

1. Incorporate a delete routine in the name-and-address-entry program.
2. Write a program to edit data in the mailing-list file. Display each item and ask if the user wants to make a change.
3. Write a program to display all data from the file for names having a specified code.

PROGRAMMER'S CORNER 8

Options in DOS Commands

Disk drives are connected to the Apple through a disk controller card. Each disk controller card has room for two disk drives, which are numbered 1 and 2 and may be inserted into one of seven slots. These slots are numbered 1 to 7. The OPEN statement may select the slot and drive with the S and D options. If you have only one disk controller and only one disk drive then these options default to your slot and drive. If you have two drives on one disk controller then you must use the D option to access drive 2. If you have more than one disk controller then to access any disk controller other than the one currently active you must use the S option.

There is a further option that allows us to number each mini floppy disk in the range from 1 to 254. This is to make it possible to coordinate data and protect data from unauthorized use. Each disk may be assigned a volume number. The assignment can only be made through the INIT command. See Appendix B for a description of INIT.

```
INIT HELLO PROGRAM ,V7
```

will perform the conventional initialization process and in addition assign the number 7 to the disk. If we use a volume number for our programs or files then we must match the volume number of the disk. In the absence of a ",V" option DOS selects 254 as the volume number.

We can open a file with a statement such as

```
199  PRINT D$;"OPEN NEW FILE ,S5 ,D2 ,V7"
```

The file NEW FILE will be opened in slot 5 in drive 2 on a disk numbered 7. All further disk access will be to slot S and drive D until we change the drive and/or slot using these options in another disk-access statement. If the disk in slot 5 drive 2 is not volume 7 then DOS reports a VOLUME MISMATCH error. To make the file random-access, simply include the length option in the OPEN statement. The order in which the options appear in the OPEN statement is not significant.

.... Protecting a File

We can protect our programs and files with LOCK.

```
199  PRINT D$;"LOCK NEW FILE ,S5 ,D2 ,V7"
```

will prevent someone from deleting our file or from writing to it. The UNLOCK statement is used to reverse the action of LOCK. A locked file appears in the CATALOG with an asterisk to the left of the file name.

.... APPEND and POSITION

APPEND performs the OPEN function and prepares the file so that a WRITE statement begins writing at the end of the file. This saves reading all data just to position to the end of a sequential file.

POSITION ,Rx positions the file pointer to the xth field ahead of the current position. This is done by counting carriage-return characters in the file. POSITION cannot be used to skip past empty data in a random-access file. POSITION cannot move the pointer closer to the first field in the file. OPEN sets the file pointer at the beginning of the file.

.... Byte

For random-access files we may position at a particular byte of a record with the byte option

```
199  PRINT D$;"READ NEW FILE ,R1Ø ,B6"
```

sets up to record 1Ø so that the next INPUT request will read from byte number 6. Thus this instruction will skip over bytes Ø through 5.

.... MON and NOMON

The MON command allows us to MONitor file activity.

```
299  PRINT D$;"MON C ,I ,O"
```

causes all Commands, Input, and Output concerning files to be displayed

on the screen. To see only the commands use

```
299  PRINT D$;"MON C"
```

Any display turned on by the MON command can be turned off with the corresponding NOMONitor command.

Chapter 9

Hi-Res Graphics

We saw in Chapter 2 and Programmer's Corner 2 that we could convert the screen into a graphics area containing up to 192Ø little blocks. We could select from among 16 colors for each block.

High-resolution graphics on an Apple II or Apple II Plus provides more dots and fewer colors. We can easily plot dots in a graphics area that is 28Ø by 16Ø. That gives us 448ØØ dots. We will also have 4 lines at the bottom of the screen for standard text display. If we do not require those 4 text lines, we can create a graphics screen that is 28Ø by 192. POKE −163Ø2,Ø converts the mixed text/Hi-Res screen to full-screen Hi-Res. That gives us 5376Ø dots.

9-1...Introduction to Hi-Res Graphics in Applesoft

There are just four commands for controlling the Hi-Res screen: HGR, HCOLOR, HPLOT, and TEXT. Let's look at them all before we attempt to write our first program.

.... The Hi-Res Graphics Screen

The statement

```
1ØØ  HGR
```

prepares the Apple for Hi-Res graphics work. When this statement is exe-

cuted, the screen is divided into 2 parts. The top part is organized into 280 columns and 160 rows. This gives us the 44800 dots mentioned earlier. The remainder of the screen is reserved for 4 lines of regular text display. Each dot in the graphics area is identified by its column and row. The columns are numbered from 0 to 279 going from left to right. The rows are numbered from 0 to 159 going from top to bottom. This is not the same as the conventional rectangular coordinate system widely used in mathematics, but this difference presents no great obstacle. The dot in the upper left corner is labeled (0,0). The dot in the lower right corner is labeled (279,159). The Apple is restored to the conventional full-text screen with the TEXT statement

```
290  TEXT
```

The Hi-Res graphics screen has a profound effect upon the memory of the computer. In fact the Hi-Res graphics screen is a "window" into memory itself. What we see on the video is memory. This memory is located in the 8K from address 8192 to address 16383. (You can convince yourself of this by using POKE to place values directly into memory in that address range.) Right away this means that our programs must not grow in length past address 8191. If that happens, executing HGR will "wipe out" the portion of our program that falls in that range. If we are using a disk system, we must have at least 32K of memory. Otherwise, HGR will "wipe out" DOS itself.

.... Hi-Res Colors

Even if we are working with a black-and-white monitor, we will have to pay attention to color. The HGR statement presents us with an all-black screen and does not change the plotting color, so we cannot be certain just what the plotting color is. The HCOLOR= statement is used to select a value in the range 0 to 7. The corresponding color names are shown in Figure 9-1.

0	Black1	4	Black2
1	Green	5	Orange
2	Violet	6	Blue
3	White1	7	White2

Figure 9-1. Hi-Res color values. (Colors depend on the TV.)

The Applesoft statement

```
120  HCOLOR= 3
```

will set the Hi-Res graphics color to white1.

.... Plotting Dots

```
150  HPLOT X,Y
```

will plot a dot at (X,Y) on the high-resolution graphics screen. The color used will be the last Hi-Res color set in an HCOLOR= statement. See Program 9-1.

```
100  HGR
110  HCOLOR= 3
120  HPLOT 0,0
130  HPLOT 0,159
140  HPLOT 279,159
150  HPLOT 279,0
```

Program 9-1. Plot dots in the four corners.

Program 9-1 will place a white dot in each of the four corners of the graphics screen. At least that is what we would think. It turns out that there are some limits on what colors may be plotted where. A white dot plotted in an odd-numbered column is really green (that is, if we select white1), and a white dot plotted in an even-numbered column is really violet. For white2 an odd column produces orange, while an even column plots as blue. Don't despair; we can easily produce white dots by plotting two dots next to each other. Now our dots will be wider, but they will be white. So we might want to fix our program as shown in Program 9-2.

```
100  HGR
110  HCOLOR= 3
120  HPLOT 0,0 : HPLOT 1,0
130  HPLOT 0,159 : HPLOT 1,159
140  HPLOT 279,159 : HPLOT 278,159
150  HPLOT 279,0 : HPLOT 278,0
```

Program 9-2. Plot dots in the four corners (white this time).

.... Lines in Hi-Res

There is no HLIN or VLIN statement in Hi-Res graphics. Instead we have a powerful extension of the HPLOT statement.

```
100  HPLOT  X,Y TO X1,Y1
```

plots a line going from X,Y to X1,Y1. This is much more flexible than HLIN or VLIN. HPLOT . . . TO may be used for horizontal, vertical, and diagonal lines. We can easily extend Program 9-1 to place a border around the graphics screen. See Program 9-3.

```
100  HGR
110  HCOLOR= 3
120  HPLOT 0,0 TO 0,159
130  HPLOT 0,159 TO 279,159
140  HPLOT 279,159 TO 279,0
150  HPLOT 279,0 TO 0,0
```

Program 9-3. HPLOTting a border on the Hi-Res screen.

It is often desirable to have a border around a graphics display. So, let's write a subroutine to do that right now. We could write the four statements 12Ø–15Ø from Program 9-3 as a single line by using three colons to create a multiple statement. However, HPLOT allows us to include multiple TOs. See Program 9-4.

```
598  REM  * PLOT A BORDER
600  HPLOT 0,0 TO 0,159 TO 279,159 TO 279,0 TO 0,0
610  RETURN
```

Program 9-4. Subroutine to plot a border.

This border has a violet left edge and a green right edge. This is the same problem we encountered earlier plotting dots. The left edge is an even column (column Ø), and the right edge is odd (column 279). We can fix that by making those edges two dots wide. See line 61Ø of Program 9-5.

```
598  REM  * PLOT A BORDER
600  HPLOT 0,0 TO 0,159 TO 279,159 TO 279,0 TO 0,0
610  HPLOT 1,0 TO 1,159 : HPLOT 278,0 TO 278,159
620  RETURN
```

Program 9-5. Program 9-4 with color-correction plotting.

Line 61Ø of Program 9-5 plots a second vertical line on the left and right edges of the screen. When plotting dots and lines one dot wide, violet and blue appear only in even-numbered columns, and green and orange appear only in odd-numbered columns. We eliminate all of this grief by making our plots two dots wide. From now on we can use GOSUB 6ØØ as calling for a Hi-Res border in the currently active HCOLOR=. The ability to continue plotting with multiple TOs is very useful.

So, there we have it. HGR, HCOLOR=, HPLOT, and TEXT give us tremendous power to draw figures on the Hi-Res graphics screen. When plotting white we must plot two horizontally adjacent dots to really get white. We can get the other colors in the same way.

For demonstration purposes let's write a program to display the Hi-Res colors. We need the usual HGR to prepare the graphics screen. Next, we should label the colors. This can be done by displaying the color number just beneath each vertical color bar. In order to do this we have to prepare

the 4-line text window. HOME clears the entire 24-line text screen in Hi-Res graphics. But now the cursor is hidden in the upper left corner of the text screen. Only the last 4 lines of the text screen are visible below the upper 160 lines of the Hi-Res graphics screen. In Lo-Res mode HOME clears only the bottom 4 lines. VTAB 21 places the cursor at the first line of the window. Alternatively, we could set the top of the text window with

```
POKE 34,20
```

and then code the HOME statement. We get a white border by setting HCOLOR= to 3 and calling our border-plotting subroutine at 600. Next, for each color, we simply calculate some attractive spacing and plot vertical bars. See Program 9-6.

```
   90  REM  * DISPLAY HI-RES COLORS
   100  HGR
   106 :
   108  REM  * PREPARE TEXT WINDOW
   110  HOME : VTAB 21
   116 :
   118  REM  * WHITE BORDER
   120  HCOLOR= 3 : GOSUB 600
   166 :
   168  REM  * COLORS 0 THRU 7
   170  PRINT "  ";
   180  FOR C = 0 TO 7
   185  PRINT "   ";C;
   190  HCOLOR= C
*  200 B = 28 * C + 34
*  210  FOR X = 1 TO 8
   220  HPLOT X + B,5 TO X + B,154
   230  NEXT X
   250  NEXT C
   300  PRINT : PRINT : PRINT  TAB( 8);
   320  PRINT "HI RES COLORS ON APPLE ";
*  330  PRINT  CHR$ (93); CHR$ (91);
   590  END
   596 :
   598  REM  * PLOT A BORDER
   600  HPLOT 0,0 TO 0,159 TO 279,159 TO 279,0 TO 0,0
   610  HPLOT 1,0 TO 1,159 : HPLOT 278,0 TO 278,159
   620  RETURN
```

Program 9-6. Display Apple Hi-Res colors.

Line 200 simply calculates a starting point for each color bar. Line 210 sets up a FOR loop to plot bars eight dots wide. Line 330 uses the CHR$ function to display the square brackets in "Apple][(II)." CHR$(93) produces], while CHR$(91) gives us [.

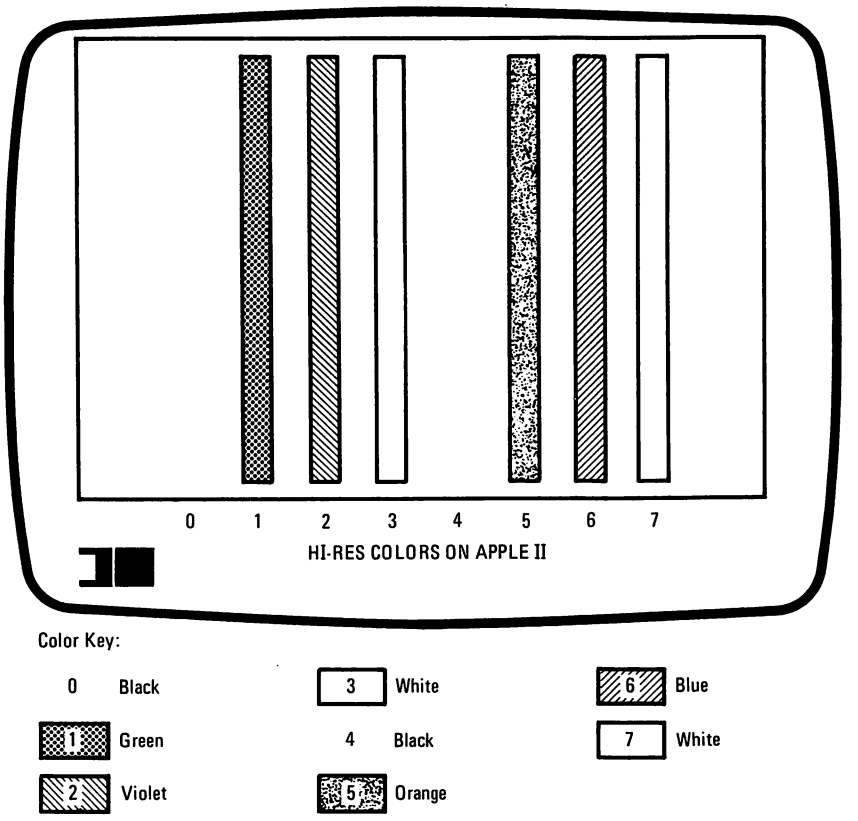


Figure 9-2. Execution of Program 9-6.

Now that we have the fundamentals we can work on making a drawing on the screen. We can simply code a series of HPLOT statements to draw lines and dots on the screen. Then, to add a line, we add an HPLOT statement. To remove a line we remove an HPLOT statement. Using this method each new drawing is a new program.

A different approach is to write a little routine that HPLOTs lines using data stored in DATA statements. We can completely specify any line and any HCOLOR with five numbers—one for the color and two for each end of the line. To plot a single dot, simply make both ends of the line the same point. This makes the plotting routine very simple indeed. Once we perfect it, we may use it for any other drawing by simply changing the DATA. It is easy to terminate plotting by looking for a color value of −1. Program 9-7 is a very compact subroutine to plot drawings from data.

```
196 :
198  REM  * VECTOR PLOTTING ROUTINE
200  READ C,X,Y,X1,Y1
210  IF C =  - 1 THEN 290
220  HCOLOR= C
230  HPLOT X,Y TO X1,Y1
240  GOTO 200
290  RETURN
```

Program 9-7. Plot drawings from data.

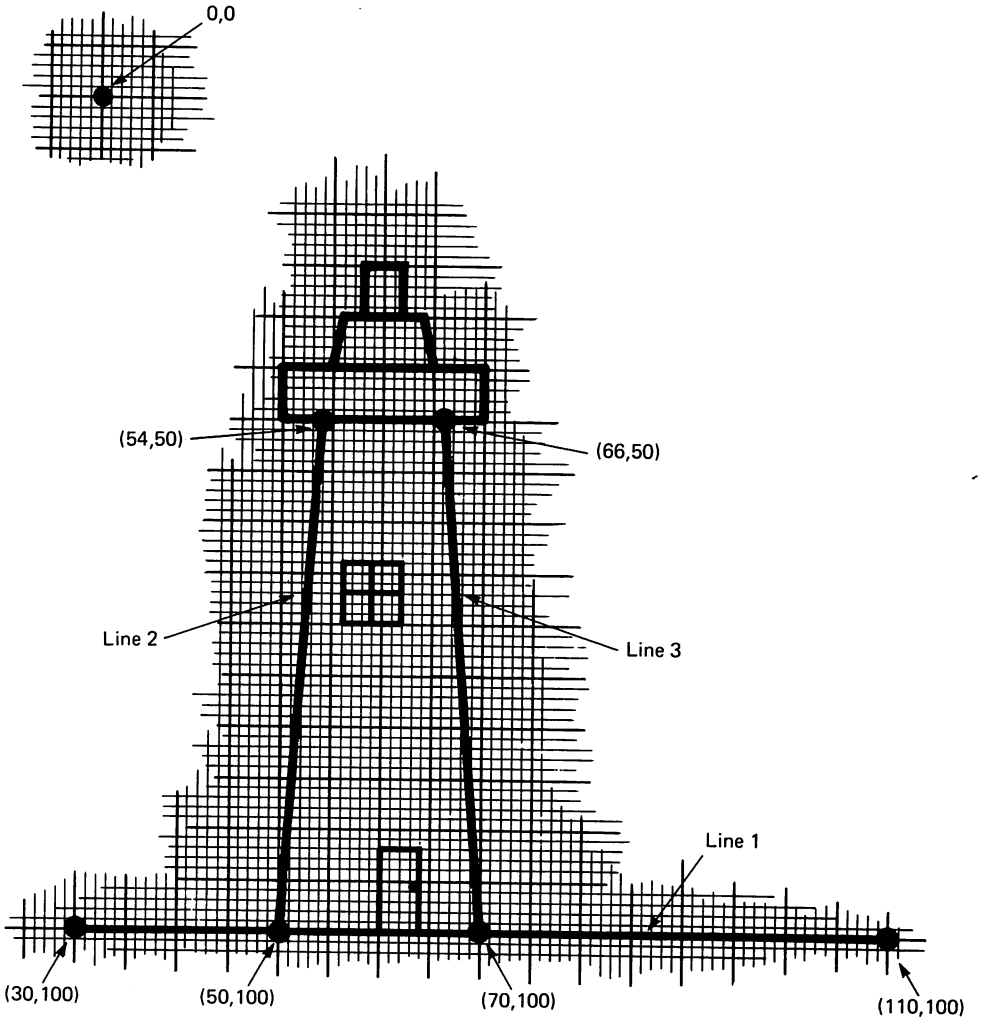


Figure 9-3. Drawing of a lighthouse on cross-section paper.

Program 9-7 is surprisingly simple. It is always very nice to come upon a short routine that does so much. This routine assumes that the Hi-Res graphics screen has been prepared. The real work in this drawing business is producing the data.

Just for fun let's draw a lighthouse. We should do the drawing on cross-section paper so that we can easily read the X,Y coordinates for each end of each straight line in the drawing. See Figure 9-3 on page 177. The first three lines are numbered as examples in Figure 9-3. Line 1 is represented by the data 3,3Ø,1ØØ,11Ø,1ØØ. Line 2 is represented by the data 3,5Ø,1ØØ,54,5Ø. Line 3 is represented by the data 3,7Ø,1ØØ,66,5Ø. In a similar fashion we obtain the rest of the data shown in Program 9-8a. When we have so much data in a program like this, it is a good idea to insert REMs to separate the data into sensible groups.

```
100  HGR : POKE  - 16302,0
110  GOSUB 200
130  END
196 :
198  REM  * VECTOR PLOTTING ROUTINE
200  READ C,X,Y,X1,Y1
210  IF C =  - 1 THEN 290
220  HCOLOR= C
230  HPLOT X,Y TO X1,Y1
240  GOTO 200
290  RETURN
996 :
998  REM  * VECTOR DATA
999  REM  * THE TOWER
1000  DATA  3,30,100,110,100
1005  DATA  3,50,100,54,50
1010  DATA  3,70,100,66,50
1013  REM  * TOP OF TOWER
1015  DATA  3,50,50,70,50
1020  DATA  3,50,50,50,45
1025  DATA  3,70,50,70,45
1030  DATA  3,50,45,70,45
1035  DATA  3,55,45,56,40
1040  DATA  3,65,45,64,40
1045  DATA  3,56,40,64,40
1050  DATA  3,58,40,58,35
1055  DATA  3,62,40,62,35
1060  DATA  3,58,35,62,35
1063  REM  * THE DOOR
1065  DATA  3,60,100,60,92
1070  DATA  3,60,92,64,92
1075  DATA  3,64,92,64,100
1077  DATA  3,63,96,63,96
1088  REM  * THE WINDOW
1090  DATA  3,56,70,62,70
1095  DATA  3,56,67,62,67
```

```
1100  DATA  3,56,64,62,64
1105  DATA  3,56,70,56,64
1110  DATA  3,59,70,59,64
1115  DATA  3,62,70,62,64
1990  DATA  -1,0,0,0,0
```

Program 9-8a. Draw a lighthouse using data and Hi-Res.

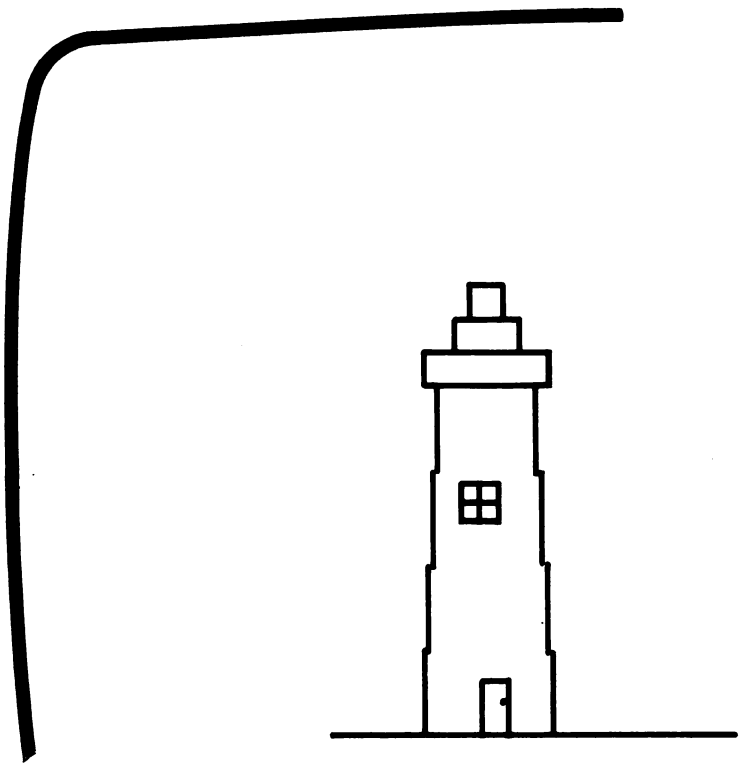


Figure 9-4. Execution of Program 9-8a.

As long as we have gone this far with the lighthouse, what with a door and a window, we really ought to have a blinking light, don't you think? One of the nice things about working with subroutines is that we can easily add new things to our programs. We can go into our main routine of Program 9-8a and insert a subroutine call to a blinking-light subroutine at line 300. Program 9-8b lists these two subroutines.

```
296 :
298  REM    * SET UP BLINKING
300  FOR I8 = 1 TO 150
310  HCOLOR= 0
330  GOSUB 400
```

```
340  HCOLOR= 3
350  GOSUB 400
360  NEXT I8
390  RETURN
396 :
398  REM  * LIGHT HERE
400  HPLOT 59,37 TO 61,37
410  HPLOT 59,38 TO 61,38
420  FOR I9 = 1 TO 500 : NEXT I9
490  RETURN
```

Program 9-8b. Blinking light for lighthouse.

This is hard to show with a figure in a book. You will have to type this one in to see it work.

There is always room for improvement. Programs 9-8a and 9-8b can draw only one lighthouse of one size at one spot on the screen. We might convert the data so that every point is calculated in terms of a single starting point. That way we will be able to move the lighthouse to any point that keeps the entire figure on the screen. Our border-drawing subroutine could be used to frame our picture. We could determine the data for many figures and save it in data files on disk. Then we will have a whole library of figures to use for later graphics applications. The possibilities are truly unlimited. We might get really spiffy and work out a way to adjust the size of the figure according to a scale factor. This last variation is probably best left for shape tables.

....SUMMARY

Just 4 Applesoft keywords open the way to very powerful Hi-Res color graphics. HGR gives us a screen with 280 columns and 160 rows. POKE −16302,0 enables an additional 32 rows at the bottom of the screen. Colors in the range from 0 to 7 are available with HCOLOR=. In order to get white we must plot the points (X,Y) and (X+1,Y). Violet and blue appear only in even-numbered columns, while green and orange may be plotted only in odd-numbered columns. HPLOT . . . TO plots single points or line segments in any orientation. We reenable the text screen with the TEXT statement. We have developed a routine that allows us to specify a drawing in terms of a collection of line segments. For each segment we need only supply the color and the endpoints.

Problems for Section 9-1 .....

The possibilities for drawing figures on the screen are literally unlimited. We can only begin to make some suggestions leading you into problems of interest. Let your imagination lead you into exciting graphics demonstrations.

1. Modify the border-plotting subroutine of Program 9-5 so that it may also be used to plot a border around the full graphics screen. Require that the calling routine set the bottom edge by setting the variable BE to either 159 or 191.
2. Adjust the data in the lighthouse-drawing program so that each set of data is calculated in terms of a fixed starting point. Using (XØ,YØ) as (3Ø,1ØØ), the first three data lines will be

```
1ØØØ  DATA  3,Ø,Ø,8Ø,Ø
1ØØ5  DATA  3,2Ø,Ø,24,-5Ø
1Ø1Ø  DATA  3,4Ø,Ø,36,-5Ø
```

Now the control routine can select a variety of starting points and draw the lighthouse anywhere on the screen with just one plotting subroutine.

9-2...Hi-Res Graphs from Formulas in Applesoft

Figures that can be described using a formula are easy to graph. There are many examples from mathematics.

.... Cartesian Coordinates

Let's develop a method for adjusting the X and Y values in the conventional Cartesian coordinate system for plotting on the Apple screen. We would like to move the (Ø,Ø) point nearer the center of the screen and alter the orientation for Y values so that they are increasing up instead of down. Suppose we specify that the point (14Ø,8Ø) on the Apple screen shall represent the point (Ø,Ø) in a Cartesian system. The X conversion is easy. We simply want to move each plotted point to the right on the screen. The Y conversion requires that we turn the graph "upside down." So the point

(X1,Y1)

in the conventional Cartesian coordinate system becomes

(140+X1,8Ø−Y1)

on the Apple Hi-Res screen.

It would be nice to plot the X and Y axes right on the screen. A very simple subroutine will do this for us. Again, here we can plot the vertical line two dots wide.

Plotting points that fit a formula is straightforward enough. For our first graphs we might do just functions. This is a good application for a DEFined function. We need a subroutine that scans all possible values for X and determines if the Y value is on the screen. If it is, then the routine should do the plotting. If not, then the routine should simply try the next X value. All of this is done in Program 9-9.

```
90  REM  * PLOT A FUNCTION
100  HGR : HOME
116 :
118  REM  * WHITE BORDER
120  HCOLOR= 3 : GOSUB 600
126 :
128  REM  * PLOT AXES
130  GOSUB 700
146 :
148  REM  * DRAW THE GRAPH
150  HCOLOR= 1
160  DEF  FN F(X) = X
170  GOSUB 200
190  END
196 :
198  REM  * PLOT A FUNCTION
200  FOR X1 =  - 138 TO 138
220 Y1 =  FN F(X1)
230 X = 140 + X1
240 Y = 80 - Y1
250  IF Y < 3 OR Y > 156 THEN 270
260  HPLOT X,Y : HPLOT X + 1,Y
270  NEXT X1
290  RETURN
596 :
598  REM  * PLOT A BORDER
600  HPLOT 0,0 TO 0,159 TO 279,159 TO 279,0 TO 0,0
610  HPLOT 1,0 TO 1,159 : HPLOT 278,0 TO 278,159
620  RETURN
696 :
698  REM  * PLOT AXES FOR GRAPHING
700  HPLOT 3,80 TO 276,80
710  HPLOT 140,3 TO 140,156
720  HPLOT 141,3 TO 141,156
790  RETURN
```

Program 9-9. Plot a function in Hi-Res.

This program is set up for the mixed text/graphics screen. We could easily convert the subroutines at lines 600 and 700 to plot for either full or part screen using an S0 value, which could be 191 for full screen and 159 for part screen. In addition we might want to move the axes so that the point (0,0) is not in the exact center. This could be done by passing (X0,Y0) to the axes-plotting subroutine as the Apple coordinates of the (0,0) point for the Cartesian graph.

.... Polar Graphs

Polar equations often produce interesting graphs. One of the reasons we don't draw many polar graphs is that they take too much tedious calculation involving trigonometric functions. We can easily produce the graphs

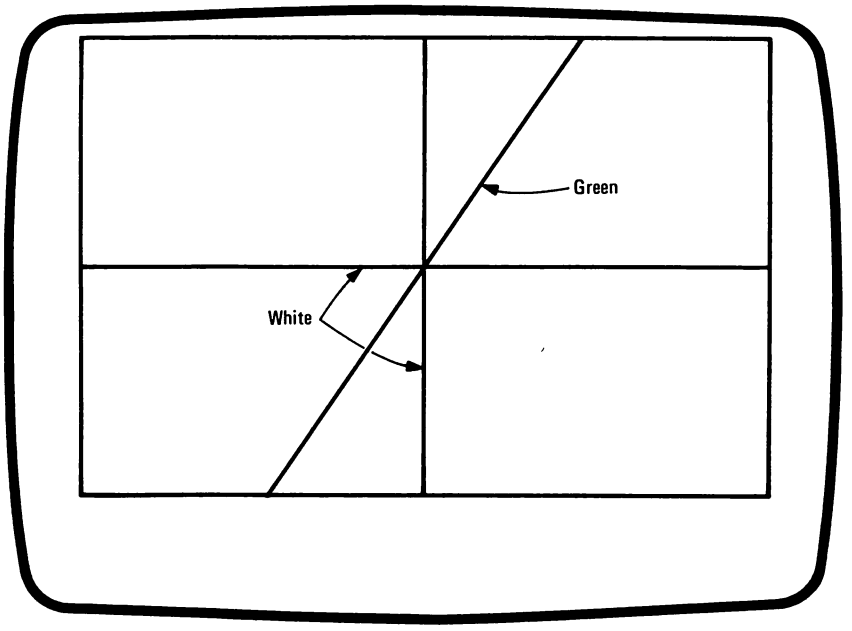


Figure 9-5. Execution of Program 9-9.

without the tedium by using Hi-Res graphics and letting Applesoft do the calculations.

We may use

R = 1 − 2cos(G)

as a sample equation. Using sines and cosines we get the X and Y coordinates as follows:

X = Rcos(G)

and

Y = Rsin(G)

where G is the central angle in radians. To obtain a full graph the central angle must sweep through a full 360 degrees or 2π. That is about 6.29. We can get about 60 points by using STEP .1 in a FOR . . . NEXT loop. Since the point (0,0) is in the corner of the Apple Hi-Res screen we need to adjust the starting point to keep the figure in view.

To make our figures as large as possible we can use POKE −16302,0 to obtain full-screen graphics. In this situation there is no text display, so after we have had a chance to examine the graph, we will need to type TEXT "in the blind" to get back the text screen and see our program. Now we have to think about adjusting the X and Y values on the conventional

Cartesian coordinate system for plotting on the Apple screen. The X conversion is easy. We simply want to move each plotted point to the right on the screen. The Y conversion requires that we turn the graph "upside down." So the point

(X9,Y9)

in the conventional Cartesian coordinate system becomes

(X+X9,Y−Y9)

on the Apple Hi-Res screen. Where the point (X,Y) defines the point on the Apple screen is where we want the Cartesian point (Ø,Ø) to be located.

It would be nice to display a polar axis right on the screen with the graph. We can easily plot a line beginning at the point (Ø,Ø) and extending to the right edge of the Apple screen. Placing the polar axis on the screen will clearly locate the graph for us.

Once we have a working program, it will be a simple matter to plug in other equations. In this way we can look at dozens of graphs in the time it would take to draw a single graph by hand. It is interesting to watch the figures as they are formed on the screen. Drawing a polar graph by hand (like typing a 1ØØ-page paper on a portable typewriter) is one of those things everybody ought to do once in his or her lifetime.

Our program divides nicely into three packages: the control routine, the polar-axis-plotting routine, and the graph-plotting routine. Let's work on them in that order.

In the control routine we set up the full graphics screen with HGR and POKE −16302, Ø. Setting the color is easy. Next we define the X and Y axes and call the polar-axis-plotting subroutine. Polar graphs plotted true size are usually very small. So we should provide a scaling factor to produce a larger graph. We define the radial scale in RS. In the actual plotting subroutine we will be arranging for the central angle to range through a full rotation of 2π. But we might like to control the step size in the control routine. Thus we set the value of ST here. Finally we call the plotting subroutine. That is all there is to it. See Program 9-1Øa.

```
100  HGR : POKE  - 16302,0
110  HCOLOR= 3
120 X = 139 : Y = 95
130  GOSUB 1000 : REM  * PLOT POLAR AXIS
140 RS = 25 : ST = .1
150  GOSUB 200 : REM  * PLOT THE GRAPH
190  END
```

Program 9-1Øa. Control routine for polar graphing.

In Program 9-1Øa line 12Ø sets the axes as close to the center of the screen as possible. Line 14Ø sets the radial scale at 25 and the step size at .1.

The easy one is the polar-axis-plotting routine. All we do is HPLOT a line from the point (X,Y) to the right edge of the screen. That takes one statement. See Program 9-1Øb.

```
996 :
998 REM  * PLOT POLAR AXIS
1000  HPLOT X,Y TO 279,Y
1030  RETURN
```

Program 9-1Øb. Draw a polar axis.

Now let's look at the actual plotting subroutine. We need to provide for the angle to sweep a full rotation. This is done with a FOR . . . NEXT loop ranging from Ø to 6.29. The number of points we want plotted may well depend on the size of the graph. We may want more points for larger graphs. So we let the calling routine establish the STep size. A large step size will not give enough points on the graph, while too small a step size will take too long to plot. We can then experiment with each new equation until we get a nice graph. Again we may plot two points next to each other to give uniform colors. See Program 9-1Øc.

```
196 :
198  REM  * PLOT POLAR GRAPH
200  FOR G = Ø TO 6.29 STEP ST
210 R1 = 1 - 2 *  COS (G)
220 R9 = RS * R1
230 X9 = R9 *  COS (G) : Y9 = R9 *  SIN (G)
240  HPLOT X + X9,Y - Y9
242  HPLOT X + X9 + 1,Y - Y9
250  NEXT G
290  RETURN
```

Program 9-1Øc. Polar-graph-plotting subroutine.

In Program 9-1Øc, the polar equation is defined in line 21Ø, the scaling factor is implemented in line 22Ø, and the Cartesian X and Y values are calculated in line 23Ø. It will be a simple matter to change the polar equation by changing line 21Ø. We must be aware that other polar equations may contain points that are off the screen. We can test for out-of-range values and skip the plotting for those points. Further, we must be alert for equations that may cause BASIC to attempt to divide by Ø. See Figure 9-6 for a trial run of this program.

Problems for Section 9-2

1. We can easily plot a circle with our polar equation plotting program using the polar equation R = 1. Do this.

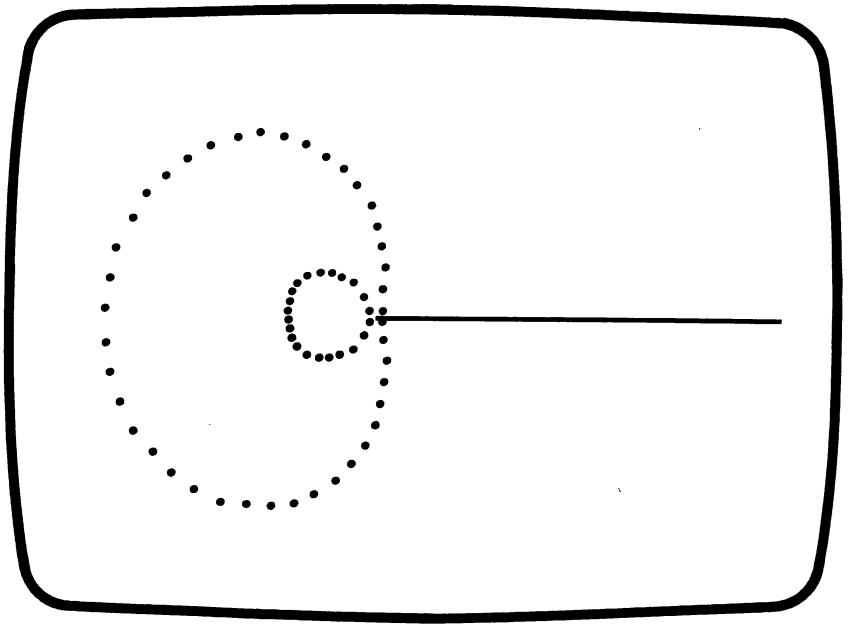


Figure 9-6. Execution of Program 9-1Ø a, b, c.

8. There are lots of interesting polar graphs. Graph any of the following:
   a. R = 1 + 2cos(G) − 3sin(G)²
   b. R = 2 + sin(3G)
   c. R = 2 + sin(2G)
   d. R = sin(G) + cos(G)

9. Many polar equations produce nice graphs, but they will cause our polar-plotting program to fail. Some points will lie off the graphics screen. Some values of G will cause division by Ø. We can easily test whether a point is on the screen between lines 23Ø and 24Ø of Program 9-1Øc. If a point is off the screen, don't plot it. If the formula we enter at line 21Ø has an indicated division then we can put in a test between lines 2ØØ and 21Ø. If the current value of G would cause such a Ø division, don't even execute line 21Ø. Adding these features will enable you to draw graphs for any of the following:
   a. Rcos(G) = 1
   b. R = 1 + Rcos(G)
   c. R = tan(G)
   d. R = 2G (make the scale 1 and make G range from −5Ø to 5Ø)
   e. R = 2/G (scale 25 and G from −1Ø to 1Ø)

PROGRAMMER'S CORNER 9

Shapes

Hi-Res graphics with shapes is marvelous. Once we place a shape definition in Apple's memory, a special set of shape keywords makes for easy shape plotting. DRAW lets us draw a shape, and XDRAW lets us draw a shape in the complement of the HCOLOR used for DRAW. ROT permits us to rotate a shape, and SCALE changes the size of shapes. Creating a shape table seems tedious at first, but once you've done a few it gets better.

Shapes are described with a sequence of plotting vectors. The first task is to draw a shape on graph paper. Suppose we want to draw a boat using shapes. First we draw the boat. See Figure 9-7(A).

There are eight kinds of arrows available. We can plot or not plot in each of the four directions. Select a starting point in the drawing of Figure 9-7(A) and trace the boat using only arrows that move up, right, down, or left and either plot or not plot. Since there are eight possible kinds of arrows they may be represented in three binary bits as shown in Table 9-1.

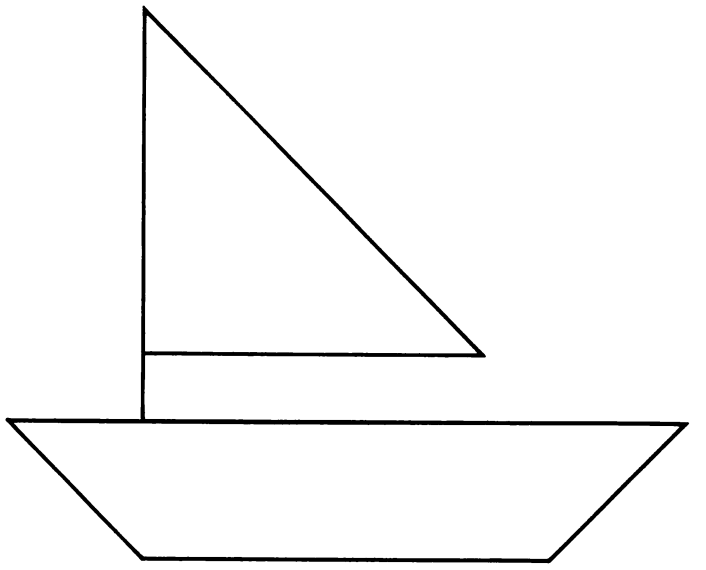
VECTOR	CODE	ACTION
↑	ØØØ	move
→	ØØ1	move
↓	Ø1Ø	move
←	Ø11	move
↑	1ØØ	plot and move
→	1Ø1	plot and move
↓	11Ø	plot and move
←	111	plot and move

Table 9-1. Shape vectors and their codes.

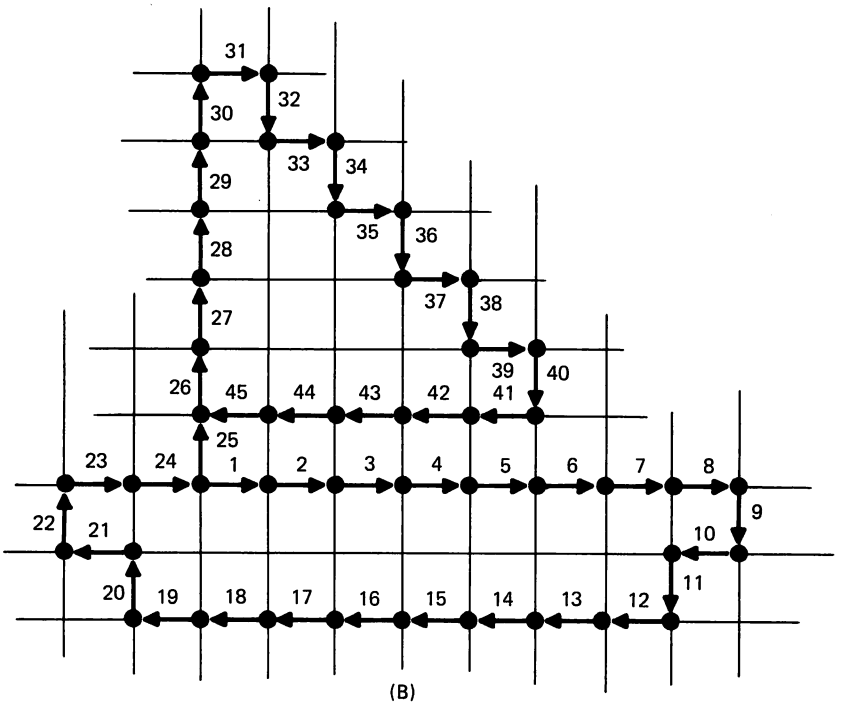
Each byte of a shape table stored in memory may store up to three vector codes, provided the third vector code is for a nonplotting vector. The vector codes are entered into the byte beginning with the low-order bits. Thus, the binary byte

Ø11ØØ111 Ø1 / 1ØØ / 111

represents the three vectors 111, 1ØØ, and Ø1 in that order from right to left. Such a byte in a shape table will cause a plot and move to the left followed by a plot and move up followed by a move to the right without plotting. Once we have a byte such as the one above, we may convert it to a decimal value that may be POKEd into memory. Or we may convert it to



(A)



(B)

Figure 9-7. Drawing a boat for a shape table.

a hex value and enter it into memory using the Apple monitor. Using decimal values makes it easy to store a shape table as data statements in a program. Shape tables are easy to edit and save in this way. Since a byte is eight bits and a plotting vector requires three bits, the third or leftmost plotting vector in a byte can only be a nonplotting vector; it cannot be the vector move up because all zeros at the end of a byte will be ignored. In fact, we could totally ignore those two bits and use each byte to store only two vector codes. The end of a shape table is signified by a byte containing all zeros. Let's unravel the vectors in our boat drawing in Fig. 9-7(A) to get a sequence of decimal values for our shape table.

Now we need to know about the structure of a shape table in memory. A shape table is segmented into two distinct parts. The beginning of a shape table contains the indexing information required to find the actual shape definitions. The rest of the shape table is made up of the shape definitions. The first byte must be the number of shape tables in the range Ø to 255. The second byte is ignored. Next, each pair of bytes is how far from the first byte of the table the corresponding shape definition begins. This is sometimes called the *offset*. The offset for the first shape table goes in the third and fourth bytes of the table. The offset for the second shape table goes in the fifth and sixth bytes. Thus for two shape tables, the first shape table will have an offset of six because the first six bytes are used for indexing information. For a shape table beginning at memory address K1, the Nth shape-table offset must be entered in memory bytes K1+2*N and K1+2*N+1.

It is very important that the Apple "know" where the shape table begins. The address where the table begins must be entered into memory at locations 232 and 233 (or E8 and E9 hex). This may also be done with POKEs. After you have some experience with shape tables and Apple memory, you will know where in memory to store shape tables. For our example we may use memory location 3ØØ hex. There are 256 bytes there that we may use for this purpose. That area of memory was left for Apple users to locate things like shape tables and machine-language routines. If we want to use that space for some other purpose or if our shape table is more than 256 bytes, then we have to look elsewhere in Apple's memory.

Another place for shape tables is just before DOS. When we boot up a disk DOS places its starting address in memory locations 115 and 116. We can find that address with the statement

```
]PRINT PEEK(116)*256 + PEEK(115)
```

Suppose we get 384ØØ and we want room for 1Ø24 bytes. We can provide the necessary space with

```
]HIMEM:37376
```

That is 384ØØ − 1Ø24. We may include the HIMEM: statement as the 1st

PT#	VECTOR (BOAT)	SHAPE	BINARY	DECIMAL
1	→	101	101101	45
2	→	101		
3	→	101	101101	45
4	→	101		
5	→	101	101101	45
6	→	101		
7	→	101	101101	45
8	→	101		
9	↓	110	111110	62
10	←	111		
11	↓	110	111110	62
12	←	111		
13	←	111	111111	63
14	←	111		
15	←	111	111111	63
16	←	111		
17	←	111	111111	36
18	←	111		
19	←	111	100111	39
20	↑	100		
21	←	111	100111	39
22	↑	100		
23	→	101	101101	45
24	→	101		

Figure 9-8. Shape table for a boat.

PT#	VECTOR (MAST)	SHAPE	BINARY	DECIMAL
25	↕	100	100100	36
26	↕	100		
27	↕	100	100100	36
28	↕	100		
29	↕	100	100100	36
30	↕	100		

PT#	VECTOR (SAIL)	SHAPE	BINARY	DECIMAL
31	↔	101	110101	53
32	↕	110		
33	↔	101	110101	53
34	↕	110		
35	↔	101	110101	53
36	↕	110		
37	↔	101	110101	53
38	↕	110		
39	↔	101	110101	53
40	↕	110		
41	↔	111	111111	63
42	↔	111		
43	↔	111	111111	63
44	↔	111		
45	↔	111	111	7
46		0		
47		0	0	0
48		0		

Figure 9-8 (concluded).

statement of an Applesoft program. Next, we have to figure out the 2 byte values to place at 232 and 233. 37376/256 comes out evenly to 146, so the low-order byte is Ø and the high-order byte is 146. There is nothing wrong with allowing a little extra space. If that gives us numbers that are convenient to work with, then so much the better.

Program 9-11 is set up to load our shape table at address 3ØØ hex. The subroutine beginning at line 9ØØ does it all. The number 3ØØ hex goes in 2 bytes in memory. The 3 is the high-order byte, and Ø is the low-order byte. See the data in line 1ØØØ of Program 9-11. This is a simple program to draw our boat in 3 different sizes at 3 different locations on the screen. With just a few minor changes you can make this program rock the boat in color.

```
1  HOME
100  GOSUB 900 : REM  * POKE SHAPE TABLE DATA
110  HGR
120  HCOLOR= 3 : GOSUB 600 : REM  * PLOT BORDER
196 :
198  REM  * PLOT SOME BOATS
200  HCOLOR= 3
210  SCALE= 1 : ROT= 0
220  DRAW 1 AT 10,10
250  SCALE= 2
260  DRAW 1 AT 30,30
300  SCALE= 5
310  DRAW 1 AT 65,65
590  END
596 :
598  REM  * PLOT A BORDER
600  HPLOT 0,0 TO 0,159 TO 279,159 TO 279,0 TO 0,0
610  HPLOT 1,0 TO 1,159 : HPLOT 278,0 TO 278,159
620  RETURN
896 :
898  REM  * POKE SHAPE TABLE
900  READ A1,A2
905  POKE 232,A1 : POKE 233,A2
910 K1 = A1 + 256 * A2
915  READ N1
920  POKE K1,N1
925 OS = 2 * N1 + 2
930  FOR I9 = 1 TO N1
935 I2 =  INT (OS / 256) : I1 = OS - 256 * I2
940  POKE K1 + 2 * I9,I1 : POKE K1 + 2 * I9 + 1,I2
945  READ P : POKE K1 + OS,P
950 OS = OS + 1
955  IF P <  > 0 THEN 945
960  NEXT I9
980  RETURN
992 :
994  REM  * BEGIN SHAPE TABLE DATA
996 :
```

```
998  REM  * STARTING ADDRESS
1000  DATA  0,3
1096 :
1098  REM  * NUMBER OF SHAPE TABLES
1100  DATA  1
1196 :
1198  REM  * FIRST SHAPE TABLE
1200  DATA  45,45,45,45,62,62,63,63,63,39,39,45
1210  DATA  36,36,36,53,53,53,53,53,63,63,7,0
```

Program 9-11. Draw a few boats with one shape.

The subroutine at line 900 may be used to POKE any collection of shape tables in memory. It does all of the necessary offset calculations for us. All we have to do is create the shapes. Line 220 simply draws shape number 1 with the starting point at 10,10 on the Hi-Res screen. The general form is

```
DRAW S AT X,Y
```

XDRAW 1 AT 10,10 would draw the same shape in the complementary color. We can create a flickering effect by repeating DRAW followed by XDRAW in a loop. We can also draw a figure and then set the HCOLOR= to the background and draw it again. This will make the figure disappear. If we then redraw the figure in the original color in a new location we can create the appearance of motion. By creating a series of figures we can produce animation.

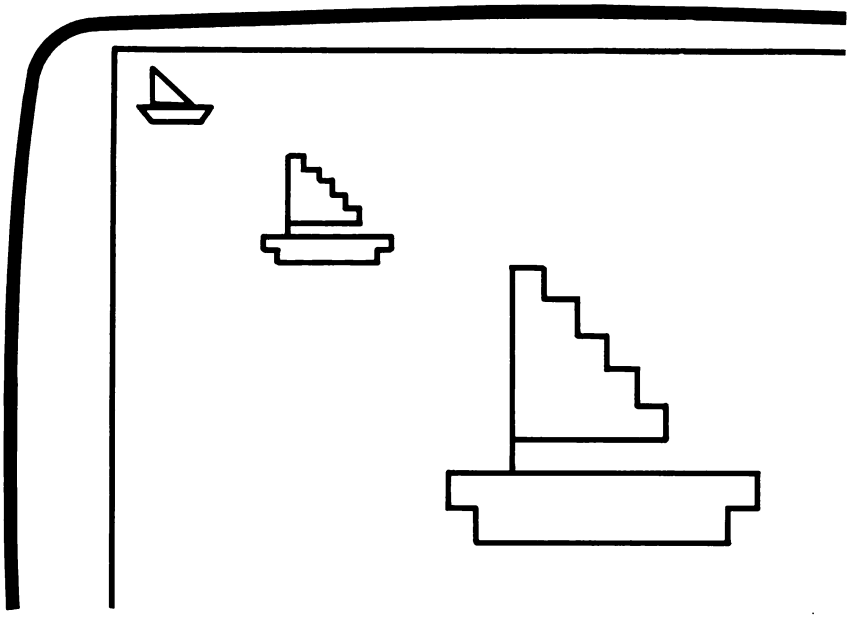


Figure 9-9. Plotting some boats using a shape table.

Line 25Ø sets the scale at 2. This means that DRAW will produce a figure on a scale of 2 to 1 compared to our original figure. ROT= may be set in the range Ø to 63 to obtain figures rotated in 64ths of a revolution. ROT= is limited by the scale. The number of values that produce unique rotations is 4 times the scale value. Thus, for SCALE = 1, we may set ROT= to Ø, 16, 32, or 48.

Plotting with shapes is subject to the same odd-even column color limitations as HPLOT. To assure that a shape is drawn in the desired HCOLOR=, we may simply DRAW at X,Y and X+1,Y.

Shape tables are enjoyable to work with, but making up shape tables by hand can be somewhat tedious. There are numerous programs and items of hardware on the market to help us with this. If you are doing a lot of work with shapes, it would be worth your while to investigate these products.

Appendix A

In the Beginning

A-1...Setting Up the Machine

Many dealers will set up and check out the new computer at the time that you purchase it. Where circumstances permit, it is a good idea to ask for a demonstration. However, little time or great distance may dictate that the buyer set up and check out his or her own machine. Meticulously follow the instructions in the manual that is included with the computer. Never plug in or unplug anything inside the Apple case unless the power is off. Failure to observe this one rule will incur significant damage to the electronics of the Apple (or any other computer).

A-2...From BASIC to BASIC

When an Apple II is turned on, a star appears in the lower left corner. To enter BASIC, simply type the letter B while holding down the key marked CTRL. Then release both keys and press the key marked RETURN. The machine responds by displaying a "greater than" symbol (>). With the Apple II Plus the screen will clear and "APPLE II" will appear with the] symbol, showing that the unit is in BASIC.

The Apple II comes with Apple Integer BASIC in ROM (Read Only Memory). This means that this language is always ready at the flick of a memory pointer. The CTRL-B-and-RETURN sequence causes the Apple II to display a "greater than" symbol (>). This signifies that you are under

the influence of Apple Integer BASIC. The Apple II Plus comes with Applesoft in ROM. Thus Applesoft is instantly available, and the CTRL-B-and-RETURN sequence is not needed. A right square bracket (]) indicates that Applesoft is the language of the moment. The "greater than" symbol (>) or the right square bracket (]) will serve as a constant reminder of which BASIC you are working with.

If you have a disk system on an Apple II, then you may gain access to Applesoft by issuing the DOS (Disk-Operating System) command "FP". Never try to run Applesoft as an Apple Integer BASIC program, even though it appears in the catalog with the "I" designation, which apparently indicates that it is an Integer BASIC program.

Should you hit the RESET key and find yourself in the monitor program, typing CTRL-C will reenter Applesoft, but it will not reconnect you to DOS. The way back is to type 3DØG ("three dee zero gee").

.... ROM Card Applesoft

The Applesoft II Firmware Card must be plugged into slot Ø at the rear of the computer. The power must be turned off to install this card (or any other card). On older cards (without the Autostart ROMs), pressing RESET will put you in the monitor with the STAR prompt. In this case, to get into Applesoft, type CØ8Ø RETURN and then CTRL-B RETURN. To get into Integer BASIC, type CØ81 RETURN and then CTRL-B RETURN. On newer cards with Autostart ROMs, pressing RESET brings back the prompt character associated with whichever BASIC you were working with. In this case, in order to change BASICs, you need the command CALL −151 to get into the monitor. Once in the monitor, issue the sequence described above. In DOS 3.1, pressing RESET with the Autostart ROMs will cause the disk drive to boot the disk. This is guaranteed to destroy your program.

Appendix B

Saving and Retrieving Programs

B-1...Tape

....Saving on Tape

You can use a cassette tape recorder to save your programs. Make sure that the cassette two-wire cable is plugged in so that the jack labeled OUT on the Apple is connected to the jack labeled MIC, MICROPHONE, or INPUT on the tape recorder. With a blank cassette in the recorder, rewind the tape, but make sure that you don't try to record your program on the tape leader. If you are not recording at the beginning of the tape it is a good idea to use the tape counter and keep a written account of what is recorded at what reading of the tape counter. Type SAVE, then start your recorder in RECORD mode. Finally, press the RETURN key on the Apple. The cursor should disappear. The computer should soon beep to signal that the beginning of the program has been sent out to the tape recorder. The next beep should signal that the program has been recorded. The time between beeps will be short for short programs and long for long programs. The cursor should reappear with the appropriate prompt character at this point. If any of these things fails to happen, hit the RESET key (gently, please), and try again. Let's hope that your DOS doesn't boot the disk. It is very important to note that the Apple has no way to determine whether or not the tape recorder is hooked up correctly or even at all. So, it is a good idea to rewind the cassette and listen to what you think you just recorded. Turn the volume down below 5Ø% (so your ears don't hurt), remove the plug labeled EAR (or whatever) and PLAY the tape. You won't

hear any Bach. What you should hear is a steady high-pitched tone for about five or ten seconds followed by a crackling sound. If you started with a blank tape, that is your program. If you are reusing an old tape, then you cannot be certain that what you are listening to is not the old program. After you have recorded a few programs, you will develop confidence in the procedure and it will become automatic for you.

Once you have a program developed to the point where you will be using it regularly, it is a good idea to make backup copies. One method for doing this consists simply of saving the same program twice on the same cassette. In addition to this, it makes sense to save a copy on a separate cassette. If your program cassette should become damaged, this second cassette will save a lot of work. If you have kept a running written record, the consequences of a lost program are kept to a minimum.

.... Retrieving Programs from Tape (LOAD)

We record our programs so that we may later retrieve them. First make sure that the two-wire cable is plugged in so that the jack labeled IN on the Apple is connected to the jack labeled EAR, EARPHONE, MON, or MONITOR on the tape recorder. Rewind the cassette to the beginning of your program. If you kept a record of the tape counter, reading this will be easy. Start playing the tape and type "LOAD" followed by RETURN at the Apple keyboard. The cursor will disappear and soon a beep will be heard. Then some time later a second beep will be heard and the cursor will reappear. The time between beeps is short for short programs and long for long programs. If you get "ERR" or "*** MEM FULL ERR" adjust the volume and/or tone on your cassette recorder and try again.

B-2...Disk

If you have a disk system, saving programs is much easier and faster than it is for cassette tapes.

.... INITializing a Disk

In order to save a program on a disk the disk must first be *initialized.* First boot up the system disk that comes with the disk drive and disk controller. Then remove it and insert a brand new blank disk. Next, type "NEW" and write a little program that you'd like to have executed every time the disk is booted up. Finally, prepare the disk for saving programs by typing "INIT PROGRAM NAME". PROGRAM NAME is a name of your choice. The disk will whirr, and the light will stay on for a couple of minutes. Now your new disk is ready to store programs for you. This initialization process must be done only once for each disk since it erases anything that is already there. After you have initialized a few disks, you won't have to think about it until it is time to initialize a new batch of disks.

.... Saving on Disk

Of course you must first write the program. Next, merely place the disk on which you want to save the program in the disk drive and type "SAVE" followed by the name you would like to use later to retrieve it. If you get an error message indicating that the disk is full, then delete any partial program that may have been saved there, get another disk, and try again. The command to erase a program from disk is "DELETE PROGRAM NAME". When all the activity dies down and the light on the disk drive goes off, type "CATALOG" to verify that your program made it to the disk.

.... Warning

It is extremely important to realize that DOS must be booted up *before* you type your program. There is no way to boot DOS without destroying your program. Should you find yourself in the unhappy situation of wanting to save a long program that you have written without booting DOS, you can still recover. Attach your cassette recorder and save your program on tape. Then boot DOS, LOAD your program from tape, and SAVE it on disk.

Appendix C

CALLS, PEEKS, and POKES

C-1...CALLs

The statement

CALL -868

causes the computer to execute the machine-language instructions that begin at the memory location whose address is -868. Certain addresses are provided that perform defined functions and then return to the active BASIC program. What about that negative address? The address is really 64668, but that value is outside the integer range of Integer BASIC. Memory has 65536 addresses from Ø to 65535. Addresses from 32768 to 64535 may also be labeled -32768 to -1. So we must subtract 65536 from 64668 to get a value that is within range. This particular CALL clears the screen from the cursor to the end of the line.

CALL -151	Puts you in the "monitor." This provides the ability to work directly with the 65Ø2 processor.
CALL -868	Clears the display screen from the cursor to the end of the current line.
CALL -912	Scrolls the text up one line on the screen.
CALL -922	Generates a line feed on the text screen.
CALL -936	Clears the screen and places the cursor in the upper left-hand corner. This position is called the HOME position.

CALL −958	Clears the screen from the cursor to the end of the page.
CALL −1994	Fills the first 2Ø lines of the page-1 text screen with @ signs. If Apple is in GR mode, then the 4Ø-by-4Ø graphics screen is cleared to all black.
CALL −1998	Fills all 24 lines of the page-1 text screen with reversed @ signs. In GR mode the 4Ø-by-4Ø graphics screen is cleared to black, and the 4 lines of text at the bottom are filled with reversed @ signs. In full-screen graphics mode, the entire 4Ø-by-48 graphics screen is cleared to black.
CALL 62454	Clears the Hi-Res screen to the last HCOLOR last HPLOTted. (Not available in Integer BASIC.)
CALL 6245Ø	Clears the Hi-Res screen to all black. (Not available in Integer BASIC.)

C-2...PEEKs and POKEs

PEEK allows us to read the data value in any byte in memory. For example:

```
PRINT PEEK(5Ø)
```

will probably produce the value 255. This indicates that the Apple is in normal display mode. A 127 there indicates a flashing display, and a 63 indicates inverse mode (dark characters on a light background).

POKE allows us to write data values into any byte in RAM. ROM BASIC is not in RAM, so we cannot write data values there. For example:

```
POKE 5Ø,63
```

will produce the inverse display mentioned above.

There are some addresses in BASIC that produce the same effect whether we use PEEK or POKE. PEEK is a function that produces a numeric value in the range Ø to 255. So, in a program we need a statement such as

```
935 P1 = PEEK(X)
```

On the other hand, POKE is a BASIC statement and may stand by itself.

....PEEKs

PEEK (−16384)	Reads 1 character from the keyboard. If the value is greater than 127, then a character has been entered. See also POKE −16368,Ø.
PEEK (−16336)	Produces a click in the Apple speaker. POKE −16336,Ø has the same effect.

PEEK (−16287)	Reads the game button at PDL(Ø). If this value is greater than 127 then the game button is being pressed. The value drops below 128 when the button is released.
PEEK (−16286)	Reads the game button at PDL(1).
PEEK (−16285)	Reads the game button at PDL(2).
PEEK (36)	Returns the horizontal position of the cursor in the range of Ø to 39.
PEEK (37)	Returns the vertical position of the cursor in the range of Ø to 23.

.... POKEs

The first 8 POKEs may be used to control the screen mode with respect to TEXT, Hi-Res, Lo-Res, and pages 1 and 2. Page 1 of Hi-Res graphics occupies from 8192 to 16383 of memory. Page 2 of Hi-Res occupies from 16384 to 24575. Page 1 of Lo-Res graphics uses memory from 1024 to 2047 and page 2 of Lo-Res graphics uses from 2048 to 3095. These POKEs are often referred to as *switches* because they switch between 2 modes. While some of these switches seem to be equivalent to BASIC keywords, there are some important differences. For example, if we switch on the Hi-Res screen with HGR, Applesoft also clears the screen to all Black1, while doing this with POKE −16297,Ø will restore the Hi-Res screen display.

Note that the switches may arranged in 4 pairs: [−163Ø3 and −163Ø4], [−163Ø1 and −163Ø2], [−16299 and −163ØØ], and [−16297 and −16298]; because each reverses the action of the other.

POKE −163Ø4,Ø	Places the Apple in Lo-Res graphics. This differs from GR. GR also clears the graphics screen to all black, but this POKE has no effect on the screen itself or the value of COLOR=.
POKE −163Ø3,Ø	Sets the Apple in text mode.
POKE −163Ø2,Ø	Enables full-screen graphics in either Hi-Res or Lo-Res.
POKE −163Ø1,Ø	Sets either graphics to mixed graphics and text. This provides the 4 lines of text at the bottom of the screen.
POKE − 163ØØ,Ø	Displays page 1 of text, Lo-Res, or text, whichever is active at the time of the POKE. This is the normal page. We don't need this unless we have previously displayed page 2.
POKE −16299,Ø	Displays page 2 of text, Lo-Res, or Hi-Res, whichever is active at the time of the POKE.
POKE −16298,Ø	Switches from Hi-Res graphics to the corresponding text page (1 or 2). However, the display on the

screen remains unchanged. This switch is needed in the event that one goes from Hi-Res graphics in Applesoft to Integer BASIC. Without this, GR might produce the previous Hi-Res screen.

POKE −16297,Ø Switches to Hi-Res graphics in the current page number.

The next 4 POKEs may be used to define the text window. If we are going to change the left margin and the window width, it is important to change the width before changing the left margin.

POKE 32,L Set the left edge of the text scroll window. L must be in the range Ø to 39.

POKE 33,W Define the width of the text scroll window. W must be in the range 1 to 4Ø.

POKE 34,T Set the top line of the text scroll window. T ranges from Ø to 23.

POKE 35,B Set the bottom line of the text scroll window. B ranges from 1 to 24.

These two POKEs are concerned with the position of the cursor.

POKE 36,H Places the cursor on the current line at the Hth character position. The sensible range for H is Ø to 1 less than the screen width.

POKE 37,V Places the cursor on line V in the range Ø to 23.

Appendix D

Index of Programs in Text

Appendix E

Solution Programs for Even-Numbered Problems

Each two-page spread should be read from top to bottom as one individual page.

Chapter 1

Problem No. 2

```
10  INPUT A,B,C,D,E
20  PRINT A + B + C + D + E

]RUN
?124.3,657,801.45,-9,81
1654.75
```

Problem No. 4

```
10  LET X = 1 / 3
20  PRINT "    X = ";X
30  PRINT "  X*3 = ";X * 3
40  PRINT "X+X+X = ";X + X + X

]RUN
    X = .333333333
  X*3 = 1
X+X+X = 1
```

```
150 T=R*10/D
160 PRINT Q;".";T
999 END

>RUN
ENTER INTEGERS TO BE DIVIDED (N,D)
?7,2
3.5
```

Chapter 2

Section 2-1

Problem No. 2

```
100  REM   CHECK AVERAGE PACKAGE
     WEIGHT FOR 180 GRAM MINIMUM
200 T1 = 0
210 C1 = 1
220  PRINT "WT ";C1;
230  INPUT WT
232  IF WT = 0 THEN  GOTO 999
```

Problem No. 6

```
10 N = 1 / 2 + 1 / 3
20 D = 1 / 3 - 1 / 4
30  PRINT N / D

]RUN
10
```

Problem No. 8

```
10  PRINT 1 * 2 * 3 * 4 * 5 * 5 * 7 * 8 * 9 * 10

]RUN
3628800
```

Problem No. 10

```
10  PRINT "ENTER NUMERATOR AND DENOMINATOR (N,D)"
20  PRINT "FOR TWO FRACTIONS TO BE MULTIPLIED"
30  PRINT
40  PRINT " FIRST FRACTION ";
50  INPUT N1,D1
60  PRINT "SECOND FRACTION ";
70  INPUT N2,D2
80  PRINT
90  PRINT "THE PRODUCT: ";N1 * N2;"/";D1 * D2

]RUN
ENTER NUMERATOR AND DENOMINATOR (N,D)
FOR TWO FRACTIONS TO BE MULTIPLIED

 FIRST FRACTION ?1,3
SECOND FRACTION ?5,7

THE PRODUCT: 5/21
```

Problem No. 12

```
100 PRINT "ENTER INTEGERS TO BE DIVIDED (N,D)"
120 INPUT N,D
130 Q=N/D
140 R=N-Q*D
```

```
235 T1 = T1 + WT
240 C1 = C1 + 1
250  IF C1 <  = 5 THEN  GOTO 220
260 AV = T1 / 5
270  IF AV < 180 THEN  GOTO 290
275  PRINT "ACCEPT THIS LOT"
280  GOTO 295
290  PRINT "REJECT THIS LOT"
295  PRINT
297  PRINT
300  GOTO 200
999  END

]RUN
WT 1?179
WT 2?182
WT 3?181
WT 4?180
WT 5?179
ACCEPT THIS LOT


WT 1?0
```

Problem No. 4

```
100  READ A,B,C,D
110  PRINT " SCORES: ";A;" ";B" ";C;" ";D
120  PRINT."AVERAGE: ";(A + B + C + D) / 4
900  DATA  100,86,71,92

]RUN
 SCORES: 100 86 71 92
AVERAGE: 87.25
```

Problem No. 6

```
100  REM  COUNT AND SUM INTEGERS
     FROM 1001 TO 2213 DIVISIBLE
     BY ELEVEN
150 C1 = 0
160 I1 = 1001
170 SM = 0
```

Problem No. 6 (continued)

```
210 C1 = C1 + 1
220 SM = SM + I1
280 I1 = I1 + 11
290  IF I1 <  = 2213 THEN  GOTO 210
310  PRINT C1;"    NUMBERS"
320  PRINT SM;" SUM"

]RUN
111    NUMBERS
178266 SUM
```

Problem No. 8

```
100  REM  DOUBLE WAGES EACH DAY FOR 30 DAYS
150 DA = 1
160 WA = .01
170 T1 = 0
198  REM
200 T1 = T1 + WA
210 W1 = WA
220 WA = WA * 2
230 DA = DA + 1
290  IF DA <  = 30 THEN  GOTO 200
300  PRINT "$ ";W1;"  $ ";T1

]RUN
$ 5368709.12  $ 10737418.2
```

Problem No. 10

```
100  REM  CALCULATE THE AMOUNT OF AN ORDER
200 BK = 4 * 10.95 * (1 - .25)
220 RC = 3 * 4.98 * (1 - .15)
240 RP = 59.95
290 T1 = BK + RC + RP
300 T = T1 * (1 - .02)
400  PRINT "AMOUNT $ ";T

]RUN
AMOUNT $ 103.38902
```

```
295  PRINT
300  PRINT TA;" TAILS"
999  END

]RUN
HTTTTHHTHHTTTHHHTHHHTHTTHTTTHHHHHHHHHTH
16 TAILS
```

Problem No. 4

```
100  REM  * ROLLING TWO DICE TEN TIMES
150 C1 = 1
200 D1 =  INT ( RND (1) * 6 + 1)
210 D2 =  INT ( RND (1) * 6 + 1)
220  PRINT D1,D2
250 C1 = C1 + 1
290  IF C1 <  = 10 THEN 200

]RUN
4             2
4             5
4             2
2             5
4             5
5             5
2             5
6             6
4             6
3             3
```

Section 2-3

Problem No. 2

```
100  REM  * COUNT ODD INTEGERS FROM 5 TO 1191
190 C1 = 0
200  FOR IN = 5 TO 1191 STEP 2
210 C1 = C1 + 1
290  NEXT IN
300  PRINT "ODD INTEGERS FROM 5 TO 1191 = ";C1

]RUN
ODD INTEGERS FROM 5 TO 1191 = 594
```

Problem No. 12

```
100  REM  G FOR GIFTS
102  REM  D FOR DAY NUMBER
104  REM  G1 FOR GIFTS THIS DAY
150 G = 0
160 D = 1
200 G1 = 0
210 G1 = G1 + 1
220 G = G + G1
230  IF G1 = D THEN  GOTO 300
240  GOTO 210
300  PRINT G1,G
305 D = D + 1
310  IF D <  = 12 THEN  GOTO 200
400  PRINT "TOTAL NUMBER OF GIFTS IS: ";G

]RUN
1               1
2               4
3               10
4               20
5               35
6               56
7               84
8               120
9               165
10              220
11              286
12              364
TOTAL NUMBER OF GIFTS IS: 364
```


Section 2-2

Problem No. 2

```
150 TA = 0
198  REM  * FLIP A COIN 39 TIMES
200 FL = 1
230  IF  RND (1) < .5 THEN 270
250  PRINT "T";
255 TA = TA + 1
260  GOTO 280
270  PRINT "H";
280 FL = FL + 1
290  IF FL <  = 39 THEN 230
```

Problem No. 4

```
100  REM  * CALCULATE WAGES FOR DOUBLING EACH DAY FOR 30 DAYS
180 WA = .01
190 T1 = .01
200  FOR DA = 2 TO 30
220 WA = WA * 2
230 T1 = T1 + WA
290  NEXT DA
300  PRINT "$ ";WA,"$ ";T1

]RUN
$ 5368709.12    $ 10737418.2
```

Problem No. 6

```
100  REM  G FOR GIFTS
102  REM  D FOR DAY NUMBER
104  REM  G1 IS GIFTS TODAY
150 G = 0
160  FOR D = 1 TO 12
190 G1 = 0
200  FOR T = 1 TO D
220 G1 = G1 + T
230  NEXT T
240 G = G + G1
300  NEXT D
400  PRINT "TOTAL NUMBER OF GIFTS IS: ";G

]RUN
TOTAL NUMBER OF GIFTS IS: 364
```

Problem No. 8

```
100  REM  * USING FOR...NEXT
150  FOR DO = 1 TO 5
198  REM  * 39 FLIPS 5 TIMES
200  FOR FL = 1 TO 39
230  IF  RND (1) < .5 THEN 270
250  PRINT "T";
260  GOTO 280
270  PRINT "H";
280  NEXT FL
295  PRINT
```

Section 2-3 *Problem No. 8 (continued)*

```
296  PRINT
300  NEXT DO
999  END

]RUN
TTTTTTHTTTHHHHTHHHHTHTTHTTHHHHTHHHTHTHT

HHHTTHHHHTHHHTTTHTTHTTHTTHTHTTTHHHHHHHT

HTTTTTHTTHTHTHTTTTTTTTHHHTHTHTTHTTTTTHTT

TTTHTTTHHTTTHTTTTTTHHTTHHTTTTTTHHTHHTHT

HTTHTHTTHTTHTTTTTHTTHHTTHTHTTHHTTTHTTTT
```

Problem No. 10

```
100  REM  * USING FOR...NEXT
150 TA = 0
198  REM  * FLIP A COIN 1000 TIMES
200  FOR FL = 1 TO 1000
230  IF  RND (1) < .5 THEN 270
250  REM
255 TA = TA + 1
260  GOTO 280
270  REM
280  NEXT FL
295  PRINT
300  PRINT TA;" TAILS"
999  END

]RUN

534 TAILS
```

Problem No. 12

```
200  PRINT "LET'S DO SOME ADDITION DRILL"
205  PRINT
210  PRINT "HOW MANY PROBLEMS DO YOU WANT";
220  INPUT N0
```


Chapter 3

Section 3-1

Problem No. 2

```
98  REM  * PLOT A ONE AND A THREE
100  GR
110  COLOR= 15
120  FOR I = 11 TO 15
130  VLIN 1,7 AT I
140  VLIN 1,7 AT I + 10
150  NEXT I
190  COLOR= 0
200  PLOT 13,4
300  PLOT 22,2
310  PLOT 23,4
320  PLOT 24,6
```

Problem No. 4

```
100  GR
105  COLOR= 15
110  FOR I = 1 TO 9
120  READ A
130  VLIN 39 - A,39 AT 3 * I - 1
140  PRINT "  ";I;
150  NEXT I
160  PRINT "  WEEKS"
170  PRINT "DOLLAR SALES IN THOUSANDS"
190  END
300  DATA  30,27,26
305  DATA  31,26,30
310  DATA  38,36,34
```

Section 3-2

Problem No. 2

```
98  REM  * DISPLAY A RANDOM DIE
100  GR
110  COLOR= 15
120 X = 0
```

```
250  PRINT
260  PRINT "ENTER RANGE OF NUMBERS DESIRED";
270  INPUT LO,HI
290 RT = 0
298  REM  * BEGIN DRILL HERE
300  FOR PR = 1 TO N0
310 N1 =  INT ( RND (1) * (HI - LO + 1) + LO)
320 N2 =  INT ( RND (1) * (HI - LO + 1) + LO)
330 SM = N1 + N2
340  PRINT
350  PRINT N1;" + ";N2;" =";
360  INPUT AN
400  IF AN = SM THEN 450
410  PRINT "NO, THAT WOULD BE ";SM
420  GOTO 480
450  PRINT "RIGHT"
460 RT = RT + 1
480  NEXT PR
500  PRINT
510  PRINT "YOU GOT ";RT;" CORRECT OUT OF ";N0
```



```
]RUN
LET'S DO SOME ADDITION DRILL

HOW MANY PROBLEMS DO YOU WANT?5

ENTER RANGE OF NUMBERS DESIRED?10,41

28 + 11 =?38
NO, THAT WOULD BE 39

10 + 19 =?29
RIGHT

18 + 22 =?40
RIGHT


30 + 29 =?39
NO, THAT WOULD BE 59

14 + 33 =?47
RIGHT

YOU GOT 3 CORRECT OUT OF 5
```

```
130 Y = 0
140  GOSUB 1000
150  COLOR= 0
200 R =  INT ( RND (1) * 6 + 1)
910  IF R = 1 THEN  GOSUB 1100
920  IF R = 2 THEN  GOSUB 1200
930  IF R = 3 THEN  GOSUB 1300
940  IF R = 4 THEN  GOSUB 1400
950  IF R = 5 THEN  GOSUB 1500
960  IF R = 6 THEN  GOSUB 1600
990  END
998  REM  * DISPLAY THE DIE BACKGROUND
1000  FOR I9 = 0 TO 4
1010  VLIN Y,Y + 6 AT X + I9
1020  NEXT I9
1090  RETURN
1098  REM  * PLOT A 'ONE'
1100  PLOT X + 2,Y + 3
1190  RETURN
1198  REM  * PLOT A 'TWO'
1200  PLOT X + 1,Y + 1
1210  PLOT X + 3,Y + 5
1290  RETURN
1298  REM  * PLOT A 'THREE'
1300  PLOT X + 1,Y + 1
1310  PLOT X + 2,Y + 3
1320  PLOT X + 3,Y + 5
1390  RETURN
1398  REM  * PLOT A 'FOUR'
1400  PLOT X + 1,Y + 1
1410  PLOT X + 1,Y + 5
1420  PLOT X + 3,Y + 1
1430  PLOT X + 3,Y + 5
1490  RETURN
1498  REM  * PLOT A 'FIVE'
1500  PLOT X + 1,Y + 1
1510  PLOT X + 1,Y + 5
1520  PLOT X + 3,Y + 1
1530  PLOT X + 3,Y + 5
1540  PLOT X + 2,Y + 3
1590  RETURN
1598  REM  * PLOT A 'SIX'
1600  PLOT X + 1,Y + 1
1610  PLOT X + 1,Y + 3
1620  PLOT X + 1,Y + 5
1630  PLOT X + 3,Y + 1
```

Section 3-2 *Problem No. 2 (continued)*

```
1640  PLOT X + 3,Y + 3
1650  PLOT X + 3,Y + 5
1690  RETURN
```

Problem No. 4

```
98  REM  * DISPLAY TWO RANDOM DICE IN THE LOWER LEFT CORNER
100  GR
108  REM  * FIRST DIE
110  COLOR= 15
120 X = 0
125 Y = 33
130  GOSUB 1000
135  COLOR= 0
140 R = 5
150  GOSUB 910
208  REM  * SECOND DIE
210  COLOR= 15
220 X = 10
230  GOSUB 1000
235  COLOR= 0
240 R = 3
250  GOSUB 910
900  END
910  IF R = 1 THEN  GOSUB 1100
920  IF R = 2 THEN  GOSUB 1200
930  IF R = 3 THEN  GOSUB 1300
940  IF R = 4 THEN  GOSUB 1400
950  IF R = 5 THEN  GOSUB 1500
960  IF R = 6 THEN  GOSUB 1600
990  RETURN
998  REM  * DISPLAY THE DIE BACKGROUND
1000  FOR I9 = 0 TO 4
1010  VLIN Y,Y + 6 AT X + I9
1020  NEXT I9
1090  RETURN
1098  REM  * PLOT A 'ONE'
1100  PLOT X + 2,Y + 3
1190  RETURN
1198  REM  * PLOT A 'TWO'
1200  PLOT X + 1,Y + 1
1210  PLOT X + 3,Y + 5
1290  RETURN
330  COLOR= 15
340  GOSUB 1000
350  COLOR= 0
360  GOSUB 910
400 R =  INT ( RND (1) * 6 + 1)
410 X = 15
430  COLOR= 15
440  GOSUB 1000
450  COLOR= 0
460  GOSUB 910
900  END
910  IF R = 1 THEN  GOSUB 1100
920  IF R = 2 THEN  GOSUB 1200
930  IF R = 3 THEN  GOSUB 1300
940  IF R = 4 THEN  GOSUB 1400
950  IF R = 5 THEN  GOSUB 1500
960  IF R = 6 THEN  GOSUB 1600
990  RETURN
998  REM  * DISPLAY THE DIE BACKGROUND
1000  FOR I9 = 0 TO 4
1010  VLIN Y,Y + 6 AT X + I9
1020  NEXT I9
1090  RETURN
1098  REM  * PLOT A 'ONE'
1100  PLOT X + 2,Y + 3
1190  RETURN
1198  REM  * PLOT A 'TWO'
1200  PLOT X + 1,Y + 1
1210  PLOT X + 3,Y + 5
1290  RETURN
1298  REM  * PLOT A 'THREE'
1300  PLOT X + 1,Y + 1
1310  PLOT X + 2,Y + 3
1320  PLOT X + 3,Y + 5
1390  RETURN
1398  REM  * PLOT A 'FOUR'
1400  PLOT X + 1,Y + 1
1410  PLOT X + 1,Y + 5
1420  PLOT X + 3,Y + 1
1430  PLOT X + 3,Y + 5
1490  RETURN
1498  REM  * PLOT A 'FIVE'
1500  PLOT X + 1,Y + 1
1510  PLOT X + 1,Y + 5
1520  PLOT X + 3,Y + 1
1530  PLOT X + 3,Y + 5
```

```
1298  REM  * PLOT A 'THREE'
1300  PLOT X + 1,Y + 1
1310  PLOT X + 2,Y + 3
1320  PLOT X + 3,Y + 5
1390  RETURN
1398  REM  * PLOT A 'FOUR'
1400  PLOT X + 1,Y + 1
1410  PLOT X + 1,Y + 5
1420  PLOT X + 3,Y + 1
1430  PLOT X + 3,Y + 5
1490  RETURN
1498  REM  * PLOT A 'FIVE'
1500  PLOT X + 1,Y + 1
1510  PLOT X + 1,Y + 5
1520  PLOT X + 3,Y + 1
1530  PLOT X + 3,Y + 5
1540  PLOT X + 2,Y + 3
1590  RETURN
1598  REM  * PLOT A 'SIX'
1600  PLOT X + 1,Y + 1
1610  PLOT X + 1,Y + 3
1620  PLOT X + 1,Y + 5
1630  PLOT X + 3,Y + 1
1640  PLOT X + 3,Y + 3
1650  PLOT X + 3,Y + 5
1690  RETURN
```


Problem No. 6

```
98  REM  * SIMULATE ROLLING DICE
100  GR
110  FOR D = 1 TO 20
120 X =  INT ( RND (1) * 35)
125 Y =  INT ( RND (1) * 33)
130 R =  INT ( RND (1) * 6 + 1)
140  COLOR= 15
150  GOSUB 1000
160  COLOR= 0
170  GOSUB 910
175  FOR T = 1 TO 50
180  NEXT T
190  GOSUB 1000
200  NEXT D
300 R =  INT ( RND (1) * 6 + 1)
310 X = 0
320 Y = 20
```

```
1540  PLOT X + 2,Y + 3
1590  RETURN
1598  REM  * PLOT A 'SIX'
1600  PLOT X + 1,Y + 1
1610  PLOT X + 1,Y + 3
1620  PLOT X + 1,Y + 5
1630  PLOT X + 3,Y + 1
1640  PLOT X + 3,Y + 3
1650  PLOT X + 3,Y + 5
1690  RETURN
```

Chapter 4

Section 4-1.1

Problem No. 2

```
98  REM  * COMPARE 360 DAYS WITH 365 FOR COMPOUND INTEREST
100 P = 10000
150  PRINT "AMOUNT","RATE %","DAYS"
200  READ DA,IN
210  IF DA = 0 THEN  END
220 I = (IN / 100) / DA
250 A = P * (1 + I) ^ DA
260  PRINT A,IN,DA
290  GOTO 200
900  DATA  360,5.5,  360,12.5
902  DATA  365,5.5,  365,12.5
990  DATA  0,0

]RUN
AMOUNT          RATE %          DAYS
10565.363       5.5             360
11331.2391      12.5            360
10565.3643      5.5             365
11331.2435      12.5            365
```

Problem No. 4

```
98  REM  * CALCULATE INTEREST IN A LOOP
100 P = 1000
150  PRINT "AMOUNT","RATE %","PERIODS"
200  READ PE,IN
210  IF PE = 0 THEN  END
```

Section 4-1.1 *Problem No. 4 (continued)*

```
220 I = (IN / 100) / PE
250  GOSUB 1100
260  PRINT A,IN,PE
290  GOTO 200
900  DATA  365,5.5,  360,5.5,  12,5.5
902  DATA  365,12.5,  360,12.5,  12,12.5
990  DATA  0,0
1098  REM  * CALCULATE INTEREST HERE
1100 A = P
1110  FOR P9 = 1 TO PE
1120 A = A * (1 + I)
1130  NEXT P9
1190  RETURN

]RUN
AMOUNT          RATE %          PERIODS
1056.53624      5.5             365
1056.53618      5.5             360
1056.40786      5.5             12
1133.1242       12.5            365
1133.12387      12.5            360
1132.41605      12.5            12
```


Section 4-1.2

Problem No. 2

```
  100 PRINT "CONVERT FAHRENHEIT TO CELSIUS"
  120 PRINT
  130 PRINT "ENTER FAHRENHEIT ";
  140 INPUT F
  145 IF F<=-500 THEN END
  150 C=(F-32)*5/9
  180 PRINT "CELSIUS TEMP IS: ";C
  190 GOTO 120

>RUN
CONVERT FAHRENHEIT TO CELSIUS

ENTER FAHRENHEIT ?32
CELSIUS TEMP IS: 0
```

```
120  PRINT "VALUE = ";F1
900  END
8998  REM  * YES-NO PROCESSOR
9000  INPUT AN$
9010  IF AN$ = "YES" THEN F1 = 1: GOTO 9090
9020  IF AN$ = "NO" THEN F1 = 0: GOTO 9090
9040  PRINT "'YES' OR 'NO' PLEASE"
9045  PRINT
9050  GOTO 9000
9090  RETURN

]RUN
QUESTION ?OK
'YES' OR 'NO' PLEASE

?YES
VALUE = 1
```

Section 5-1.2

Problem No. 2

```
  10 CALL -936
  90 REM  * COMPARE STRINGS FOR ORDER IN INTEGER BASIC
  95 DIM F$(250),S$(250)
 100 PRINT "ORDERING TWO STRINGS"
 105 PRINT
 110 INPUT " ENTER FIRST STRING? ",F$
 115 IF F$="STOP" THEN END
 120 INPUT "ENTER SECOND STRING? ",S$
 130 IF F$=S$ THEN 300
 135 IF LEN(F$)= LEN(S$) THEN 200
 140 IF LEN(F$)< LEN(S$) THEN 170
 150 FOR I= LEN(S$)+1 TO LEN(F$)
 155 S$( LEN(S$)+1)=" "
 160 NEXT I
 165 GOTO 200
 170 FOR I= LEN(F$)+1 TO LEN(S$)
 175 F$( LEN(F$)+1)=" "
 180 NEXT I
 190 GOTO 200
 200 FOR I=1 TO LEN(F$)
 210 IF ASC(F$(I,I))> ASC(S$(I,I)) THEN 400
```

```
ENTER FAHRENHEIT ?212
CELSIUS TEMP IS: 100

ENTER FAHRENHEIT ?68
CELSIUS TEMP IS: 20

ENTER FAHRENHEIT ?-500
```

Chapter 5

Section 5-1.1

Problem No. 2

```
98  REM  * FIND EARLIEST ITEM IN DATA
100  READ E$
110  PRINT E$
150  READ A$
155  IF A$ = "STOP" THEN 300
160  PRINT A$
170  IF A$ < E$ THEN E$ = A$
180  GOTO 150
300  PRINT
310  PRINT "ALPHABETICALLY FIRST:";E$
890  END
900  DATA  PENNSYLVANIA, NEW JERSEY
902  DATA  CALIFORNIA, TEXAS
904  DATA  VIRGINIA, FLORIDA
990  DATA  STOP

]RUN
PENNSYLVANIA
NEW JERSEY
CALIFORNIA
TEXAS
VIRGINIA
FLORIDA

ALPHABETICALLY FIRST: CALIFORNIA
```

Problem No. 4

```
98  REM  * YES-NO SUBROUTINE
100  PRINT "QUESTION ";
110  GOSUB 9000
220 IF ASC(F$(I,I))< ASC(S$(I,I)) THEN 500
230 NEXT I
300 PRINT F$;" EQUALS ";S$
310 GOTO 105
400 PRINT F$;" IS GREATER THAN ";S$
410 GOTO 105
500 PRINT F$;" IS LESS THAN ";S$
510 GOTO 105

>RUN
ORDERING TWO STRINGS

 ENTER FIRST STRING? WHAT'S THIS

ENTER SECOND STRING? WHAT'S WHAT

WHAT'S THIS IS LESS THAN WHAT'S WHAT

 ENTER FIRST STRING? COMPUTER

ENTER SECOND STRING? CALCULATOR

COMPUTER   IS GREATER THAN CALCULATOR

 ENTER FIRST STRING? STOP
```

Problem No. 4

```
 10 CALL -936
 80 REM    * DISPLAY THE DAYS OF THE WEEK
100 DIM A$(63)
120 A$="SUNDAY   MONDAY   TUESDAY  WEDNESDAYTHURSDAY FRIDAY
   SATURDAY "
130 FOR J=1 TO 9
140 FOR I=1 TO 7
150 I9=(I-1)*9+J
160 PRINT A$(I9,I9);"  ";
190 NEXT I
192 PRINT
195 NEXT J
200 END
```

Section 5-1.2 *Problem No. 4 (continued)*

```
>RUN
S  M  T  W  T  F  S
U  O  U  E  H  R  A
N  N  E  D  U  I  T
D  D  S  N  R  D  U
A  A  D  E  S  A  R
Y  Y  A  S  D  Y  D
      Y  D  A     A
         A  Y     Y
         Y
```

Problem No. 6

```
   10 CALL -936
   90 DIM S$(250)
  100 PRINT "  I WILL DISPLAY YOUR STRING IN REVERSE ORDER"
  110 PRINT
  120 INPUT "ENTER YOUR STRING? ",S$
  130 IF LEN(S$)>0 THEN 200
  135 PRINT "TOO SHORT"
  140 GOTO 110
  200 FOR I= LEN(S$) TO 1 STEP -1
  220 PRINT S$(I,I);
  230 NEXT I
  900 END

>RUN
  I WILL DISPLAY YOUR STRING IN REVERSE ORDER

ENTER YOUR STRING? THIS IS A NICE DAY.

.YAD ECIN A SI SIHT
```

Section 5-2.1

Problem No. 2

```
90  REM  * TEST FORMATTER
100  INPUT "TEST VALUE? ";M1
110  IF M1 = 0 THEN  END
120  GOSUB 1000
```

Problem No. 6

```
90  REM  * CONVERT FROM <$1,234.51> TO 1234.51
92  REM  * WE MAY USE GET$ TO PROCESS COMMAS ON INPUT
100  PRINT "ENTER TEST STRING? ";
105  GOSUB 2000
110  PRINT
120  IF M$ = "STOP" THEN  END
130  GOSUB 1200
140  PRINT M$;" BECOMES ";M9$
150  PRINT
160  GOTO 100
1198  REM  * REMOVE SPECIAL CHARACTERS
1200 S$ = "$<>,"
1210 M9$ = ""
1220  FOR I9 = 1 TO  LEN (M$)
1230  FOR J9 = 1 TO  LEN (S$)
1240  IF  MID$ (M$,I9,1) =  MID$ (S$,J9,1) THEN 1270
1250  NEXT J9
1260 M9$ = M9$ +  MID$ (M$,I9,1)
1270  NEXT I9
1280  IF  RIGHT$ (M$,1) = ">" THEN M9$ = "-" + M9$
1290  RETURN
1998  REM  * HANDLE INPUT USING GET$ TO ALLOW COMMAS
1999  REM    CARRIAGE RETURN IS ASC 13
2000 M$ = ""
2010  GET A$
2020  IF  ASC (A$) = 13 THEN 2090
2030  PRINT A$;
2040 M$ = M$ + A$
2050  GOTO 2010
2090  RETURN

]RUN
ENTER TEST STRING? <$1,234.50>
<$1,234.50> BECOMES -1234.50

ENTER TEST STRING? 23.41
23.41 BECOMES 23.41

ENTER TEST STRING? STOP
```

Problem No. 8

```
90  REM  * TEST FORMATTER
```

```
130  PRINT M1;" = ";D$
140  PRINT
150  GOTO 100
988 :
990  REM  * FORMAT DOLLARS AND CENTS
998  REM  * INSERT A DOLLAR SIGN
1000 M9 =  INT (M1 * 100 + .5)
1010 X$ =  STR$ (M9)
1020 D9 =  LEN (X$) - 2
1030 D$ =  LEFT$ (X$,D9) +  CHR$ (46) +  RIGHT$ (X$,2)
1040 D$ = "$" + D$
1090  RETURN

]RUN
TEST VALUE? 31.9
31.9 = $31.90

TEST VALUE? 0
```

Problem No. 4

```
80  REM  * HANDLE ZERO AMOUNT
82  REM    AND CHANGE THE EXIT FLAG
90  REM  * TEST FORMATTER
100  INPUT "TEST VALUE? ";M1
110  IF M1 =  - 9999 THEN  END
115  IF M1 = 0 THEN D$ = "0.00": GOTO 130
120  GOSUB 1000
130  PRINT M1;" = ";D$
140  PRINT
150  GOTO 100
988 :
990  REM  * FORMAT DOLLARS AND CENTS
1000 M9 =  INT (M1 * 100 + .5)
1010 X$ =  STR$ (M9)
1020 D9 =  LEN (X$) - 2
1030 D$ =  LEFT$ (X$,D9) +  CHR$ (46) +  RIGHT$ (X$,2)
1090  RETURN

]RUN
TEST VALUE? 0
0 = 0.00

TEST VALUE? 999.1
999.1 = 999.10

TEST VALUE? -9999
```

```
95 PL = 3
97  REM  * PL DETERMINES THE NUMBER OF PLACES
100  INPUT "TEST VALUE? ";M1
110  IF M1 = 0 THEN  END
120  GOSUB 1000
130  PRINT M1;" = ";D$
140  PRINT
150  GOTO 100
988 :
990  REM  * FORMAT TO ANY NUMBER OF PLACES
1000 M9 =  INT (M1 * 10 ^ PL + .5)
1010 X$ =  STR$ (M9)
1020  IF  LEN (X$) > PL THEN 1050
1030 T$ = "00000000000000"
1040 X$ =  RIGHT$ (T$,PL + 1 -  LEN (X$)) + X$
1050 D9 =  LEN (X$) - PL
1060 D$ =  LEFT$ (X$,D9) +  CHR$ (46) +  RIGHT$ (X$,PL)
1090  RETURN

]RUN
TEST VALUE? 102.01
102.01 = 102.010

TEST VALUE? .09
.09 = 0.090

TEST VALUE? 0
```

Problem No. 10

```
1  HOME
200  READ B$
210  READ A$: IF A$ = "DONE" THEN 999
220  FOR I9 = 1 TO  LEN (A$)
240 B$ =  RIGHT$ (B$,38) +  MID$ (A$,I9,1)
245  HTAB 1
250  PRINT B$;
255  FOR X9 = 1 TO 100: NEXT X9
260  NEXT I9
290  GOTO 210
900  DATA  "                                                    "
902  DATA  "THIS IS A SAMPLE SCROLLING MESSAGE.  "
904  DATA  "WE CAN EASILY CHANGE THE MESSAGE BY CHANGING THE DATA
     STATEMENTS.  "
989  DATA  "                                                    "
990  DATA  DONE
999  END
```

Chapter 6

Section 6-1.1

Problem No. 2

```
90  REM  * ENTER THE TEMPERATURES IN ARRAY W
100  FOR J = 1 TO 7
110  READ W(J)
120  NEXT J
145  REM  * SET UP INITIAL CONDITIONS
150 T = W(1)
160 H = W(1):L = W(1)
170 DH = 1:DL = 1
190 :
200  FOR J = 2 TO 7
210 T = T + W(J)
230  IF W(J) > H THEN H = W(J):DH = J
240  IF W(J) < L THEN L = W(J):DL = J
280  NEXT J
290 :
300  PRINT "AVERAGE TEMP: ";T / 7
320  PRINT "HIGHEST TEMP: ";H;" ON DAY ";DH
330  PRINT " LOWEST TEMP: ";L;" ON DAY ";DL
890 :
900  DATA  72,78,76,79,85,85,71
990  END

]RUN
AVERAGE TEMP: 78
HIGHEST TEMP: 85 ON DAY 5
 LOWEST TEMP: 71 ON DAY 7
```

Problem No. 4

```
90  REM  * DRAWING TEN NUMBERS AT RANDOM FROM AMONG TEN
92  REM  * COUNTING UNUSED DRAWS
95 :
100  FOR J = 1 TO 10
110 A(J) = 1
120  NEXT J
180 UN = 0
190 :
200  FOR J = 1 TO 10
210 R =  INT ( RND (1) * 10 + 1)
110  FOR J = 1 TO N1
120  READ A1(J)
130  NEXT J
200  READ N2
210  FOR J = 1 TO N2
220  READ A2(J)
230  NEXT J
300  FOR J = 1 TO N1
305 J3 = J
310 A3(J) = A1(J)
315  NEXT J
325  FOR J = 1 TO N2
330  FOR K = 1 TO J3
335  IF A3(K) = A2(J) THEN 365
340  NEXT K
345 J3 = J3 + 1
350 A3(J3) = A2(J)
355  GOTO 365
360  NEXT K
365  NEXT J
400  PRINT "THE COMPOSITE ARRAY:"
410  FOR J = 1 TO J3
420  PRINT A3(J);"  ";
430  NEXT J
800  END
900  DATA  3,  6, 3, 9
910  DATA  4,  2, 8, 6, 5

]RUN
THE COMPOSITE ARRAY:
6  3  9  2  8  5
```

Section 6-1.2

Problem No. 2

```
 90 REM   * ENTER THE TEMPERATURES IN ARRAY W
 95 DIM W(7)
 97 PRINT "ENTER SEVEN TEMPERATURES"
100 FOR J=1 TO 7
105 PRINT J;
110 INPUT W(J)
120 NEXT J
145 REM   * SET UP INITIAL CONDITIONS
150 T=W(1)
```

```
250  IF A(R) = 0 THEN UN = UN + 1: GOTO 210
260  PRINT "  ";R;
270 A(R) = 0
280  NEXT J
290  PRINT
295  PRINT UN;" UNUSED DRAWS"
300  END

]RUN
  3  8  6  9  4  1  7  2  5  10
17 UNUSED DRAWS
```

Problem No. 6

```
10  HOME
90  REM  * DISPLAY ELEMENTS IN ORDER AND IN REVERSE ORDER
100  DIM AR(20)
110  FOR J = 1 TO 20
120 AR(J) = 2 * J
130  NEXT J
200  PRINT "DISPLAY IN ORDER"
210  FOR J = 1 TO 20
220  PRINT AR(J);"  ";
230  NEXT J
240  PRINT : PRINT
300  PRINT "DISPLAY IN REVERSE ORDER"
310  FOR J = 20 TO 1 STEP  - 1
320  PRINT AR(J);"  ";
330  NEXT J
800  END

]RUN
DISPLAY IN ORDER
2  4  6  8  10  12  14  16  18  20  22  24  26  28  30  32  34  36  38  40

DISPLAY IN REVERSE ORDER
40  38  36  34  32  30  28  26  24  22  20  18  16  14  12  10  8  6  4  2
```

Problem No. 8

```
10  HOME
90  REM  * DISPLAY COMPOSITE OF TWO ARRAYS
95  DIM A1(15),A2(15),A3(30)
100  READ N1
```

```
160 H=W(1):L=W(1)
170 DH=1:DL=1
190 REM
200 FOR J=2 TO 7
210 T=T+W(J)
230 IF W(J)<=H THEN 240
232 H=W(J):DH=J
240 IF W(J)>=L THEN 280
242 L=W(J):DL=J
280 NEXT J
290 REM
300 PRINT "AVERAGE TEMP: ";T/7
320 PRINT "HIGHEST TEMP: ";H;" ON DAY ";DH
330 PRINT " LOWEST TEMP: ";L;" ON DAY ";DL
990 END

>RUN
ENTER SEVEN TEMPERATURES
1?77
2?79
3?79
4?81
5?77
6?72
7?66
AVERAGE TEMP: 75
HIGHEST TEMP: 81 ON DAY 4
 LOWEST TEMP: 66 ON DAY 7
```

Problem No. 4

```
 90 REM   * DRAWING FIVE NUMBERS AT RANDOM FROM AMONG
    TEN
 92 REM   * COUNTING UNUSED DRAWS
 95 DIM A(10)
100 FOR J=1 TO 10
110 A(J)=1
120 NEXT J
180 UN=0
190 REM
200 FOR J=1 TO 10
210 R= RND (10)+1
250 IF A(R)<>0 THEN 260
255 UN=UN+1: GOTO 210
260 PRINT "  ";R;
```

Section 6-1.2 *Problem No. 4 (continued)*

```
  270 A(R)=0
  280 NEXT J
  290 PRINT
  295 PRINT UN;" UNUSED DRAWS"
  300 END

>RUN
  6  9  2  4  7  10  1  3  5  8
37 UNUSED DRAWS
```

Problem No. 6

```
   10 CALL -936
   90 REM   * DISPLAY ELEMENTS IN ORDER AND IN REVERSE ORDER
  100 DIM AR(20)
  110 FOR J=1 TO 20
  120 AR(J)=2*J
  130 NEXT J
  200 PRINT "DISPLAY IN ORDER"
  210 FOR J=1 TO 20
  220 PRINT AR(J);"  ";
  230 NEXT J
  240 PRINT : PRINT
  300 PRINT "DISPLAY IN REVERSE ORDER"
  310 FOR J=20 TO 1 STEP -1
  320 PRINT AR(J);"  ";
  330 NEXT J
  800 END

>RUN
DISPLAY IN ORDER
2  4  6  8  10  12  14  16  18  20  22  24  26  28  30  32  34  36  38  40

DISPLAY IN REVERSE ORDER
40  38  36  34  32  30  28  26  24  22  20  18  16  14  12  10  8  6  4  2
```

Problem No. 8

```
   10 CALL -936
   90 REM   * DISPLAY ALL POSSIBLE PAIRS FROM TWO ARRAYS
   95 DIM A1(15),A2(15),A3(40)
```

Section 6-2

Problem No. 2

```
90  REM  * FIND MAXIMUM TEMP
92  REM      FOR EACH DAY
95  DIM TE(7,4)
100  FOR DA = 1 TO 7
110  FOR RE = 1 TO 3
120  READ TE(DA,RE)
130  NEXT RE
140  NEXT DA
150  GOSUB 2000
155 :
170  PRINT " MAXIMUM"
180  PRINT "DAY TEMP"
200  FOR DA = 1 TO 7
210  PRINT DA;"    ";TE(DA,4)
220  NEXT DA
230  PRINT
900  END
980 :
1000  DATA  76,79,75, 72,77,76
1020  DATA  74,79,81, 75,80,83
1040  DATA  80,77,70, 68,65,65
1060  DATA  65,67,76
1996 :
1998  REM  * FIND MAXIMUM TEMPERATURES HERE
2000  FOR DA = 1 TO 7
2010 TE(DA,4) = TE(DA,1)
2020  NEXT DA
2050  FOR DA = 1 TO 7
2060  FOR RE = 1 TO 3
2070  IF TE(DA,RE) > TE(DA,4) THEN TE(DA,4) = TE(DA,RE)
2080  NEXT RE
2090  NEXT DA
2095  RETURN

]RUN
 MAXIMUM
DAY TEMP
1   79
2   77
3   81
4   83
5   80
```

```
100 INPUT "HOW MANY NUMBERS IN FIRST ARRAY ",N1
110 FOR J=1 TO N1
120 PRINT J;: INPUT A1(J)
130 NEXT J
190 PRINT
200 INPUT "HOW MANY NUMBERS IN SECOND ARRAY ",N2
210 FOR J=1 TO N2
220 PRINT J;: INPUT A2(J)
230 NEXT J
300 FOR J=1 TO N1
305 J3=J
310 A3(J)=A1(J)
315 NEXT J
325 FOR J=1 TO N2
330 FOR K=1 TO J3
335 IF A3(K)=A2(J) THEN 365
340 NEXT K
345 J3=J3+1
350 A3(J3)=A2(J)
355 GOTO 365
360 NEXT K
365 NEXT J
400 PRINT "THE COMPOSITE ARRAY:"
410 FOR J=1 TO J3
420 PRINT A3(J);"  ";
430 NEXT J
800 END

>RUN
HOW MANY NUMBERS IN FIRST ARRAY ?3

1?5
2?91
3?23

HOW MANY NUMBERS IN SECOND ARRAY ?4

1?6
2?4
3?23
4?11
THE COMPOSITE ARRAY:
5  91  23  6  4  11
6  68
7  76
```

Problem No. 4

```
90  REM  * FILLING ARRAYS WITH RANDOM NUMBERS
95  DIM R1(4,5),R2(4,5),S(4,5)
100  FOR R = 1 TO 4: FOR C = 1 TO 5
120 R1(R,C) =  INT ( RND (1) * 398 + 101)
130  NEXT C: NEXT R
200  FOR R = 1 TO 4: FOR C = 1 TO 5
220 R2(R,C) =  INT ( RND (1) * 398 + 101)
230  NEXT C: NEXT R
296 :
300  PRINT "       FIRST ARRAY"
310  FOR R = 1 TO 4
320  FOR C = 1 TO 5
330  PRINT R1(R,C);"  ";
340  NEXT C
350  PRINT
360  NEXT R
370  PRINT
396 :
400  PRINT "      SECOND ARRAY"
410  FOR R = 1 TO 4
420  FOR C = 1 TO 5
430  PRINT R2(R,C);"  ";
440  NEXT C
450  PRINT
460  NEXT R
470  PRINT
496 :
498  REM  * ENTER SUMS HERE
500  FOR R = 1 TO 4: FOR C = 1 TO 5
520 S(R,C) = R1(R,C) + R2(R,C)
530  NEXT C: NEXT R
596 :
600  PRINT "           SUMS"
610  FOR R = 1 TO 4
620  FOR C = 1 TO 5
630  PRINT S(R,C);"  ";
640  NEXT C
650  PRINT
660  NEXT R
```

Section 6-2 *Problem No. 4 (continued)*

```
]RUN
      FIRST ARRAY
216  380  273  368  221
196  373  118  298  212
398  402  345  134  377
379  203  154  259  274

     SECOND ARRAY
122  167  130  240  497
418  291  131  490  392
263  329  190  349  369
271  110  291  334  416

          SUMS
338  547  403  608  718
614  664  249  788  604
661  731  535  483  746
650  313  445  593  690
```


Section 6-3

Problem No. 2

```
1  REM  * CHANGES IN COMPUTER RESPONSE SUBROUTINE FOR MORE RANDOM SELECTION
2 :
4  REM  * FIRST DEL 5000,5015
6  REM  * THEN ENTER THE FOLLOWING LINES
5000 ST =  INT ( RND (1) * N0 + 1)
5002  FOR I9 = ST TO N0
5004  IF  LEFT$ (NA$(I9),1) =  RIGHT$ (PE$(I9),1) AND AV(I9) = 1 THEN 5050
5006  NEXT I9
5010  FOR I9 = 1 TO ST
5012  IF  LEFT$ (NA$(I9),1) =  RIGHT$ (PE$(I9),1) AND AV(I9) = 1 THEN 5050
5014  NEXT I9
```

Problem No. 4

```
90  REM  * FIND THE ALPHABETICALLY FIRST NAME
95  DIM WE$(7)
100  GOSUB 900
196 :
```

```
INPUT AN INTEGER?817263549
8  1  7  2  6  3  5  4  9

INPUT AN INTEGER?0
```

Problem No. 4

```
  100 PRINT "INPUT AN INTEGER";
  110 INPUT N
  120 IF N=0 THEN END
  130 FOR E=4 TO 0 STEP -1
  140 T=10 ^ E
  150 I=N/T
  160 PRINT I;"  ";
  170 R=N-I*T
  180 N=R
  190 NEXT E
  200 PRINT : PRINT
  210 GOTO 100

>RUN
INPUT AN INTEGER?28931
2  8  9  3  1

INPUT AN INTEGER?0
```

Problem No. 6

```
   90 REM  * REVERSE THE DIGITS OF PRIME NUMBERS
  100 FOR X=100 TO 900 STEP 200
  110 FOR Y=1 TO 99 STEP 2
  120 N=X+Y:NU=N
  130 FOR Z=3 TO N STEP 2
  140 IF Z*Z>N THEN 170
  150 IF N/Z*Z=N THEN 370
  160 NEXT Z
  170 GOSUB 900
  200 FOR Z=3 TO RE STEP 2
  210 IF Z*Z>RE THEN 300
  220 IF RE/Z*Z=RE THEN 370
  230 NEXT Z
  300 PRINT NU,
  370 NEXT Y
  380 NEXT X
```

```
200 SM$ = WE$(1):PO = 1
210  FOR I9 = 2 TO 7
220  IF WE$(I9) < SM$ THEN SM$ = WE$(I9):PO = I9
230  NEXT I9
240  PRINT "ALPHABETICALLY FIRST = ";SM$
250  PRINT "    IN POSITION NUMBER: ";PO
800  END
896 :
898  REM  * READ WEEKDAY NAMES
900  FOR I9 = 1 TO 7
910  READ WE$(I9)
960  NEXT I9
990  RETURN
996 :
1000  DATA  SUNDAY, MONDAY, TUESDAY
1010  DATA  WEDNESDAY, THURSDAY, FRIDAY
1020  DATA  SATURDAY

]RUN
ALPHABETICALLY FIRST = FRIDAY
   IN POSITION NUMBER: 6
```


Chapter 7

Section 7-1

Problem No. 2

```
100  PRINT "INPUT AN INTEGER";
110  INPUT N
120  IF N = 0 THEN  END
130  FOR E = 8 TO 0 STEP  - 1
140 T = 10 ^ E
150 I =  INT (N / T)
160  PRINT I;"  ";
170 R =  INT (N - I * T + .5)
180 N = R
190  NEXT E
200  PRINT : PRINT
210  GOTO 100

]RUN
INPUT AN INTEGER?31345
0  0  0  0  3  1  3  4  5
```

```
590 END
895 REM
898 REM  * REVERSE HERE
900 T=0:RE=0:E1=-1
910 FOR E=4 TO 0 STEP -1
920 EX=10 ^ E
930 I=N/EX
940 T=T+I
950 IF T=0 THEN 980
960 E1=E1+1
970 RE=RE+I*10 ^ E1
975 N=N-I*EX
980 NEXT E
990 RETURN

]RUN
101     107     113     131     149
151     157     167     179     181
191     199     311     313     337
347     353     359     373     383
389     701     709     727     733
739     743     751     757     761
769     787     797     907     919
929     937     941     953     967
971     983     991
```

Section 7-2

Problem No. 2

```
100 REM * CONVERT DECIMAL TO BINARY
110 DIM A(16)
120 FOR J=1 TO 16
130 A(J)=0
140 NEXT J
200 INPUT "ENTER AN INTEGER",I
210 IF I<=0 THEN 999
296 REM
298 REM  * LOAD THE ARRAY
300 FOR J=16 TO 1 STEP -1
310 A(J)=I MOD 2
320 I=I/2
360 NEXT J
396 REM
398 REM  * DISPLAY RESULTS
```

Section 7-2 *Problem No. 2 (continued)*

```
  400 T1=0
  402 FOR J=1 TO 16
  404 T1=T1+A(J)
  406 IF T1=0 THEN 420
  410 PRINT A(J);
  420 NEXT J
  455 PRINT : PRINT
  460 GOTO 120
  999 END

>RUN
ENTER AN INTEGER?129

10000001

ENTER AN INTEGER?32766

1111111111111110

ENTER AN INTEGER?0
```


Problem No. 4

```
100  REM  * DEVELOP HEX INPUT/OUTPUT
130 H$ = "0123456789ABCDEF"
140  GOSUB 400: REM  * REQUEST & VERIFY
150  PRINT
160  PRINT N$;"   ";NU
190  GOTO 140
396 :
398  REM  * REQUEST & CALL VERIFY
400  PRINT : INPUT "HEX NUMBER? ";N$
410 L =  LEN (N$)
420  IF L = 0 THEN  END
430  IF L < 5 THEN 440
432  PRINT "TOO MANY DIGITS"
434  GOTO 400
440  GOSUB 700
450  IF FL = 0 THEN 490
460  PRINT "BAD FORMAT": GOTO 400
490  RETURN
696 :
```

```
450  PRINT "TOO SMALL"
460  GOTO 200
500  PRINT "YOU GOT IT ***"
510  END

]RUN
  I WILL THINK OF A NUMBER BETWEEN 1 AND
 N.  HOW LARGE WOULD YOU LIKE
N TO BE?91

YOUR GUESS: 45
HIGHER

YOUR GUESS: 66
LOWER

YOUR GUESS: 55
LOWER

YOUR GUESS: 50
HIGHER

YOUR GUESS: 52
YOU GOT IT ***
```

Problem No. 4

```
10  HOME
110  GOSUB 1100
120  GOSUB 1200
130  GOSUB 1300
140  GOSUB 1400
150  GOSUB 1500
200  PRINT : PRINT "MONTHLY PAYMENT = $ ";P$
210 PA =  VAL (P$) * N1
220  GOSUB 1500
230  PRINT " TOTAL PAYMENTS = $ ";P$
240 PA =  VAL (P$) - P
250  GOSUB 1500
260  PRINT " TOTAL INTEREST = $ ";P$
900  END
1096 :
1098  REM  * GET INTEREST RATE
1100  PRINT "ANNUAL INTEREST RATE (%) ";
1110  INPUT I
```

```
698  REM  * VERIFY HEX STRING
700 FL = 0: REM  * GOOD INPUT
705 NU = 0
710  FOR J = 1 TO L
720  FOR K = 1 TO 16
730  IF  MID$ (H$,K,1) =  MID$ (N$,J,1) THEN 760
740  NEXT K
750 FL = 1: REM  * BAD INPUT
755  GOTO 790
760 NU = NU + (K - 1) * 16 ^ (L - J)
770  NEXT J
790  RETURN

]RUN
HEX NUMBER? F0F0

F0F0   61680

HEX NUMBER?
```


Section 7-3

Problems of General Interest

Problem No. 2

```
90  HOME
100  PRINT "  I WILL THINK OF A NUMBER BETWEEN 1 AND N.  ";
110  PRINT "HOW LARGE WOULD YOU LIKE"
120  PRINT "N TO BE";
130  INPUT N
140  IF N < 1 THEN : PRINT "TOO SMALL": GOTO 110
150 N1 =  INT ( RND (1) * N + 1)
200  PRINT
205  INPUT "YOUR GUESS: ";G
210  IF G < 1 THEN 450
220  IF G > N THEN 400
230  IF G = N1 THEN 500
240  IF G < N1 THEN 300
250  PRINT "LOWER"
260  GOTO 200
300  PRINT "HIGHER"
310  GOTO 200
400  PRINT "THAT'S BIGGER THAN YOUR LIMIT"
410  GOTO 200
1120  IF I > 1 THEN 1140
1125  PRINT "PERCENTAGE PLEASE"
1130  PRINT : GOTO 1100
1140 I1 = I / 100
1150 I1 = I1 / 12
1188  REM  * EXIT  WITH  MONTHLY RATE IN I1
1190  RETURN
1196 :
1198  REM  * GET PRINCIPLE
1200  PRINT "     ENTER PRINCIPLE ($) ";
1210  INPUT P
1290  RETURN
1296 :
1298  REM  * GET NUMBER OF YEARS
1300  PRINT "          NUMBER OF YEARS ";
1310  INPUT N
1320 N1 = N * 12
1388  REM  * EXIT WITH NUMBER OF PAYMENTS IN N1
1390  RETURN
1396 :
1398  REM  * COMPUTE MONTHLY PAYMENT
1400 X = (1 + I1) ^ N1
1410 PA = (P * I1 * X) / (X - 1)
1490  RETURN
1496 :
1498  REM  * FORMAT PAYMENT
1500 X =  INT (PA * 100 + .5)
1510 P$ =  STR$ (X)
1520 Y =  LEN (P$) - 2
1530 P$ =  LEFT$ (P$,Y) + "." +  RIGHT$ (P$,2)
1590  RETURN

]RUN
ANNUAL INTEREST RATE (%) ?17
ENTER PRINCIPLE ($) ?50000
    NUMBER OF YEARS ?30

MONTHLY PAYMENT = $ 712.84
 TOTAL PAYMENTS = $ 256622.40
 TOTAL INTEREST = $ 206622.40
```

Math-Oriented Problems

Problem No. 2

```
50  REM  * EUCLID'S ALGORITHM
100  INPUT " FIRST NUMBER? ";N1
```

Math-Oriented Problems *Problem No. 2 (continued)*

```
105  IF N1 = 0 THEN  END
110  INPUT "SECOND NUMBER? ";N2
150 Q =  INT (N1 / N2)
160 R = N1 - N2 * Q
170  IF R = 0 THEN 200
175 N1 = N2
180 N2 = R
190  GOTO 150
200  PRINT "GREATEST COMMON FACTOR: ";N2
210  PRINT
220  GOTO 100

]RUN
 FIRST NUMBER? 1001
SECOND NUMBER? 1300
GREATEST COMMON FACTOR: 13

 FIRST NUMBER? 0
```

Problem No. 4

```
10  HOME
50  REM  * FIND PERFECT NUMBERS
100  DIM FA(50)
110 C1 = 0
200  FOR NU = 2 TO 32767 STEP 2
210 SU = 1
220 F1 = 1
230 FA(F1) = 1
250  FOR IN = 2 TO  SQR (NU)
260 X = NU / IN
270  IF X <  >  INT (X) THEN 330
280 FA(F1 + 1) = IN
290 FA(F1 + 2) = X
300 F1 = F1 + 2
310 SU = SU + IN + X
320  IF SU > NU THEN 420
330  NEXT IN
340  IF SU <  > NU THEN 420
350  PRINT NU;" IS PERFECT"
360  FOR I9 = 1 TO F1
370  PRINT FA(I9);"  ";
```

```
15              20              25
18              24              30
20              21              29
```

Problem No. 8

```
10  HOME
50  REM  * FIND PI FROM A SEQUENCE
100 PI = 2
110  FOR I9 = 1 TO 1500 STEP 4
120 PI = PI + 16 / ((I9) * (I9 + 2) * (I9 + 4))
140  NEXT I9
150  PRINT "APPROXIMATE VALUE: ";PI
900  END

]RUN
APPROXIMATE VALUE: 3.14159176
```

Chapter 8

Section 8-3

Problem No. 2

```
5 D$ =  CHR$ (4)
10 F$ = "PLACES"
20  DIM NA$(300)
30  GOSUB 8000: REM  * READ NAMES ARRAY
95  HOME
100  GOSUB 12000: REM  * EDIT NAMES SPELLING
140  GOSUB 8500: REM  * REWRITE THE NAMES FILE
190  END
7996 :
7998  REM  * READ NAMES FILE
8000  PRINT D$;"OPEN";F$
8005  PRINT D$;"READ";F$
8010  INPUT N0
8030  FOR I9 = 1 TO N0
8040  INPUT NA$(I9)
8050  NEXT I9
8060  PRINT D$;"CLOSE";F$
8090  RETURN
8496 :
8498  REM  * UPDATE NAMES FILE
```

```
380  NEXT I9
390  PRINT : PRINT
400 C1 = C1 + 1
410  IF C1 = 4 THEN  END
420  NEXT NU
900  END

]RUN
6 IS PERFECT
1  2  3

28 IS PERFECT
1  2  14  4  7

496 IS PERFECT
1. 2  248  4  124  8  62  16  31

8128 IS PERFECT
1  2  4064  4  2032  8  1016  16  508  32  254  64  127
```

Problem No. 6

```
10  HOME
50  REM  * FIND PYTHAGOREAN TRIPLES
100  FOR L1 = 1 TO 25
120  FOR L2 = L1 + 1 TO 25
130 X = L1 * L1 + L2 * L2
140  FOR HY = L2 + 1 TO 50
150 X1 = HY * HY
155  IF X1 > X THEN 180
160  IF X1 < X THEN 170
165  PRINT L1,L2,HY
170  NEXT HY
180  NEXT L2
190  NEXT L1

]RUN
3              4              5
5              12             13
6              8              10
7              24             25
8              15             17
9              12             15
10             24             26
12             16             20
```



```
8500  PRINT D$;"OPEN";F$
8510  PRINT D$;"WRITE";F$
8520  PRINT N0
8530  FOR I9 = 1 TO N0
8535  PRINT NA$(I9)
8540  NEXT I9
8580  PRINT D$;"CLOSE"
8590  RETURN
11996 :
11998  REM  * EDIT PLACE NAMES SPELLING
12000  PRINT "EDITING PLACE NAMES."
12005  PRINT
12010  INPUT "FIX NAME? ";X$
12015  IF X$ = "DONE" THEN 12090
12020  FOR I9 = 1 TO N0
12025  IF NA$(I9) = X$ THEN 12040
12030  NEXT I9
12035  PRINT "NOT FOUND": GOTO 12005
12040  INPUT "NEW PLACE? ";X$
12045  FOR J9 = 1 TO N0
12050  IF X$ = NA$(J9) THEN 12070
12055  NEXT J9
12060 NA$(I9) = X$
12065  GOTO 12005
12070  PRINT "DUPLICATE NAME - REENTER"
12075  GOTO 12005
12090  RETURN
```

Problem No. 4

```
1  REM  * CHANGES IN COMPUTER RESPONSE SUBROUTINE FOR MORE
   RANDOM SELECTION
2 :
4  REM  * FIRST DEL 5000,5015
6  REM  * THEN ENTER THE FOLLOWING LINES
5000 ST =  INT ( RND (1) * N0 + 1)
5002  FOR I9 = ST TO N0
5004  IF  LEFT$ (NA$(I9),1) =  RIGHT$ (PE$(I9),1) AND AV(I9) = 1
     THEN 5050
5006  NEXT I9
5010  FOR I9 = 1 TO ST
5012  IF  LEFT$ (NA$(I9),1) =  RIGHT$ (PE$(I9),1) AND AV(I9) = 1
     THEN 5050
5014  NEXT I9
```

Section 8-5

Problem No. 2

```
9  REM  *  ID => ENTRY IDENTIFICATION NUMBER
10  REM  *  NS => NEW SPACE
11  REM  *  DS => DELETED SPACE
30 D$ =  CHR$ (4)
50 L0 = 128
60 F$ = " FIRST FILE"
70  DIM LA$(9),LE(9),DA$(9)
95 :
98  REM  * MAILING LIST ENTRY EDITOR
100  GOSUB 1000: REM  * READ DATA LABELS
110  GOSUB 900: REM  * READ AVAILABLE SPACE PARAMETERS
120  GOSUB 3000: REM  * REQUEST ID
125  IF ID = 0 THEN  END
130  GOSUB 3100: REM  * READ AND EDIT ENTRY
140  GOSUB 600: REM  * WRITE ENTRY TO FILE
150  GOTO 120: REM  * DO IT AGAIN
595 :
598  REM   * WRITE ENTRY
600  PRINT D$;"OPEN";F$;",L";L0
610  PRINT D$;"WRITE";F$;",R";ID
620  FOR I9 = 1 TO N0
630  PRINT DA$(I9)
640  NEXT I9
650  PRINT D$;"CLOSE";F$
690  RETURN
895 :
898  REM   * READ AVAILABLE SPACE
900  PRINT D$;"OPEN";F$
910  PRINT D$;"READ";F$
920  INPUT NS
930  INPUT DS
940  PRINT D$;"CLOSE";F$
990  RETURN
996 :
998  REM   * READ DATA LABELS AND LIMITS
1000  READ N0
1010  FOR X9 = 1 TO N0
1020  READ LA$(X9),LE(X9)
1030  NEXT X9
1090  RETURN
1996 :
1998  REM  * DATA LABEL & LIMITS
3195  IF  LEFT$ (AN$,1) = "N" THEN 3205
3200  PRINT "Y OR N";: GOTO 3185
3205  INPUT "    : ";DA$(I9)
3210  IF  LEN (DA$(I9)) <  = LE(I9) THEN 3260
3220  PRINT "TOO LONG": GOTO 3205
3260  NEXT I9: GOTO 3290
3280  PRINT D$;"CLOSE";F$
3290  RETURN
```

Chapter 9

Section 9-1

Problem No. 2

```
100  HGR : POKE  - 16302,0
105 X0 = 100:Y0 = 100
110  GOSUB 200
120  GOSUB 300
130  END
196 :
198  REM  * VECTOR PLOTTING ROUTINE
200  READ C,X,Y,X1,Y1
210  IF C =  - 1 THEN 290
220  HCOLOR= C
230  HPLOT X + X0,Y + Y0 TO X1 + X0,Y1 + Y0
240  GOTO 200
290  RETURN
296 :
298  REM   * SET UP BLINKING
300  FOR I8 = 1 TO 150
310  HCOLOR= 0
330  GOSUB 400
340  HCOLOR= 3
350  GOSUB 400
360  NEXT I8
390  RETURN
396 :
398  REM  * LIGHT HERE
400  HPLOT 29 + X0, - 63 + Y0 TO 30 + X0, - 63 + Y0
410  HPLOT 29 + X0, - 62 + Y0 TO 30 + X0, - 62 + Y0
420  FOR I9 = 1 TO 500: NEXT I9
490  RETURN
996 :
998  REM  * VECTOR DATA
```

```
2000  DATA  9
2005  DATA  ID #,    4
2010  DATA  CODE,    5
2015  DATA  LAST,   20
2020  DATA  FRST,   20
2025  DATA  ADDR,   30
2030  DATA  CITY,   16
2035  DATA  STAT,    2
2040  DATA  "ZIP ",  5
2045  DATA  PHON,   17
2996 :
2998  REM  * REQUEST ID
3000  PRINT
3010  INPUT "ID #: ";ID
3020  IF ID < NS AND ID >  = 0 THEN 3090
3030  PRINT "NON-EXISTENT ID": GOTO 3000
3090  RETURN
3096 :
3098  REM  * READ THE ENTRY IF IT IS REAL
3100  PRINT D$;"OPEN";F$;",L";L0
3110  PRINT D$;"READ";F$;",R";ID
3120  INPUT X9: IF X9 = ID THEN 3140
3125  PRINT ID;" HAS BEEN DELETED"
3130 E1 = 0: GOTO 3280
3140 DA$(1) =  STR$ (X9)
3145  FOR I9 = 2 TO N0
3150  INPUT DA$(I9)
3155  NEXT I9
3160  PRINT D$;"CLOSE";F$
3165  HOME
3170  PRINT LA$(1);"  ";DA$(1)
3175  FOR I9 = 2 TO N0
3180  PRINT LA$(I9);": ";DA$(I9);
3185  PRINT  TAB( 25);"    OK";: INPUT AN$
3190  IF  LEFT$ (AN$,1) = "Y" THEN 3260
```



```
999  REM  * THE TOWER
1000  DATA  3,0,0,80,0
1005  DATA  3,20,0,24,-50
1010  DATA  3,40,0,36,-50
1013  REM  * TOP OF TOWER
1015  DATA  3,20,-50,40,-50
1020  DATA  3,20,-50,20,-55
1025  DATA  3,40,-50,40,-55
1030  DATA  3,20,-55,40,-55
1035  DATA  3,25,-55,26,-60
1040  DATA  3,35,-55,34,-60
1045  DATA  3,26,-60,34,-60
1050  DATA  3,28,-60,28,-65
1055  DATA  3,32,-60,32,-65
1060  DATA  3,28,-65,32,-65
1063  REM  * THE DOOR
1065  DATA  3,30,0,30,-8
1070  DATA  3,30,-8,34,-8
1075  DATA  3,34,-8,34,0
1077  DATA  3,33,-4,33,-4
1088  REM  * THE WINDOW
1090  DATA  3,26,-30,32,-30
1095  DATA  3,26,-33,32,-33
1100  DATA  3,26,-36,32,-36
1105  DATA  3,26,-30,26,-36
1110  DATA  3,29,-30,29,-36
1115  DATA  3,32,-30,32,-36
1990  DATA  -1,0,0,0,0
```

Section 9-2

Problem No. 2

```
50  REM  * SIMPLY ENTER PROGRAM 9-10A,B,C
51  REM  * AND TYPE LINE 210 ACCORDING TO THE FORMULAS
```

Bibliography

Apple Computer, Inc., has put out many valuable publications. Some of them are included as appropriate with various hardware and software items.

Applesoft: BASIC Programming Reference Manual. 1978.

The Applesoft Tutorial. 1979.

Apple Software Bank, Contributed Programs. Volumes 1 and 2, 1978.

Apple Software Bank, Contributed Programs. Volumes 3–5, 1978.

Apple II *BASIC Programming Manual* (Integer BASIC). 1978.

Apple II *Reference Manual, Hardware and Software.* 1979.

Apple II: The DOS Manual, Disk Operating System. 198Ø. (Includes 16-sector-diskette information.)

DOS version 3.2: Disk Operating System Instructional and Reference Manual. 1979.

Other publications devoted to the Apple II and Apple II Plus computers:

Computer Station's Programmer's Guide to the Apple II. Computer Station, 1978. (All material edited from Apple manuals listed above.)

Dougherty, William E. *The Apple II Monitor Peeled.* 1979.

Poole, Lon. With Martin McNiff and Steven Cook. *Apple II User's Guide.* Osborne, McGraw-Hill, 1981.

The following periodicals carry articles, features, and reviews of the Apple computers and peripherals:

Byte
Compute
Creative Computing
Interface Age
kilobaud Microcomputing
Nibble
On Computing
Personal Computing
SOFTLINE
SOFTALK

Index

5626-5
$12.95

Basic APPLE™ BASIC

JAMES S. COAN

This book is a complete guide to Applesoft BASIC and takes the reader from beginning concepts, such as entering data and obtaining output, and planning programs, to more advanced topics, such as numeric and string arrays, and sequential and random access files. Alternative techniques for programming in Apple Integer BASIC are also provided. Both low-resolution and high-resolution graphics are discussed.

This best-selling author follows his successful approach of beginning with short, simple programs that are gradually expanded to form complex programs that illustrate creative problem solving. Over 80 distinct useful programs are included, and all can be found easily with the convenient program index.

Programmer's Corner sections at the end of each chapter focus on special Apple II features or advanced programming ideas, including special screen editing, full-screen graphics, advanced file handling, shape tables, and more. Appendixes include getting started, loading and saving programs, commonly used PEEKs, POKEs, and CALLs, and answers to all even-numbered exercises.

Also by the same author...

Basic FORTRAN

Write meaningful FORTRAN programs immediately. This step-by-step approach to learning FORTRAN programming begins with short, complete programs that are developed into longer, more comprehensive ones. Over 80 program examples are included. #5168-9, paper, 248 pages.

Advanced BASIC: Applications and Problems

Designed for programmers who want to extend their expertise in BASIC. Offers advanced techniques and applications, including coordinate geometry, area, sequences and series, polynomials, graphing, simulations, and games. #5855-1, paper, #5856-X, cloth, 192 pages.

Programming in Applesoft BASIC

JAMES S. COAN AND SCOTT BANKS

This self-instructional software package offers a powerful, hands-on approach to computer literacy for the beginning programmer on the Apple II computer. Included are 2 disks and an Activities Book containing programming exercises and suggested solutions for 12 disk-based lessons. The package is designed for the user to have complete control of his or her learning pace. #14009 (includes text and 2 Apple II diskettes).

HAYDEN BOOK COMPANY, INC.
Rochelle Park, New Jersey

ISBN 0-8104-5626-5